Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1994

MEBuilder : an object-oriented lesson authoring system for procedural skills

Galvin, Thomas P.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/30938

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MEBUILDER:
AN OBJECT-ORIENTED LESSON AUTHORING SYSTEM FOR PROCEDURAL SKILLS

by

Thomas P. Galvin

September 1994

Thesis Advisor: Neil C. Rowe

Approved for public release; distribution is unlimited.

Thesis
G14215

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
MEBUILDER: An Object-Oriented Lesson Authoring System for Procedural Skills (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Galvin, Thomas Patrick

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many military applications for intelligent-tutoring systems focus on the training of procedural skills. However, while there have been many successful research efforts into developing tutoring systems for specific applications, the question of developing general-purpose ones is still open. Specifically unsolved is how a lesson-authoring system, a program written to help a novice write computerized lessons, can be made both general purpose and easy to use.

MEBuilder is a prototype lesson-authoring system which employs an object-oriented approach to solving this problem. MEBuilder combines automated object, task, and lesson modeling tools with a library management system to allow teachers to develop simulation-based procedural trainers on nearly any subject. Teachers create reusable objects which have a fixed and well-defined behavior. Then by using the power of means-ends analysis, MEBuilder helps the teacher build entire tasks with these objects in just one step. With these tasks, teachers use MEBuilder's workbook structure to create a lesson containing several exercises. At each step, MEBuilder's automatic error and consistency checking reduces time spent on testing and debugging. MEBuilder's library manager ensures object and task reusability. This thesis explains MEBuilder's design, data structures, and interfaces. It also presents experimental results which support MEBuilder's methods as being more efficient and authoring systems using traditional computer-aided instruction (CAI) techniques.

14. SUBJECT TERMS	15. NUMBER OF PAGES
artificial intelligence, intelligent computer-aided instruction, intelligent tutoring systems, lesson authoring system, object-oriented design	228
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

MEBUILDER: AN OBJECT-ORIENTED LESSON AUTHORING SYSTEM FOR PROCEDURAL SKILLS

by

Thomas P. Galvin
Captain, United States Army
B.S., Carnegie-Mellon University, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1994

Author: [redacted]

Thomas P. Galvin

Approved By: [redacted]

Neil C. Rowe, Thesis Advisor

[redacted]

Timothy J. Shimeall, Second Reader

[redacted]

Ted Lewis, Chairman,
Department of Computer Science

Thesis
G1415
C.2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

TABLE OF CONTENTS

I. INTRODUCTION 1
A. THE DESIRE FOR HANDS-ON STYLE TRAINING BY COMPUTER 1
B. OBJECT-ORIENTED MODELING AS AN ANSWER TO THE NEED 2
C. MEBUILDER -- AN OBJECT-ORIENTED LESSON AUTHORING SYSTEM 3
D. CONTENTS OF THIS THESIS 4

II. SURVEY OF RELATED WORK IN LESSON AUTHORING 7
A. ELEMENTS OF A GOOD LESSON-AUTHORING SYSTEM 7
B. TYPES OF LESSON-AUTHORING SYSTEMS 9
C. OTHER ISSUES REGARDING LESSON-AUTHORING 10
D. METUTOR -- A MEANS-ENDS BASED INTELLIGENT TUTORING SYSTEM 11

III. TUTORING-SYSTEM VIRTUAL WORLDS AND OBJECT-MODELING TECHNIQUES 13
A. OBJECT-MODELING TECHNIQUES 13
B. USE OF OBJECT-MODELING TECHNIQUES IN METUTOR LESSONS 14
C. OTHER BENEFITS TO USING OBJECT MODELING 16
D. PITFALLS TO USING OBJECT-MODELING TECHNIQUES 17
E. TOOLS NEEDED TO ADD OBJECT MODELING TO AN AUTHORING SYSTEM 18
F. SUMMARY 22

IV. AN INTRODUCTION TO THE MEBUILDER SYSTEM 23
A. MEBUILDER'S TOP-LEVEL DESIGN AND PHILOSOPHY 23
B. MEBUILDER MAIN MODULE -- "MEBuilder" 26
C. MEBUILDER'S LIBRARY MODULE -- "MEBuildLIB" 27
D. MEBUILDER'S CLASS DEFINITION MODULE -- "MEBuildCLS" 28
E. MEBUILDER'S TASK DEFINITION MODULE -- "MEBuildTSK" 31
F. MEBUILDER'S LESSON DEFINITION MODULE -- "MEBuildLES" 36
G. MEBUILDER'S LESSON COMPILER -- "MEBuildCMP" 37

V. TRANSLATING AN MEBUILDER LESSON TO AN METUTOR LESSON 39
A. HIGH-LEVEL DESCRIPTION OF THE DESIGN CHANGES IN METUTOR 39
B. CONCEPT OF THE TRANSLATION PROCESS 41
C. GENERATION OF THE RECOMMENDED CLAUSES 42
D. GENERATION OF THE PRECONDITION CLAUSES 42
E. GENERATION OF THE POSTCONDITION CLAUSES 43
F. GENERATION OF THE RANDCHANGE CLAUSES 44

VI. EXPERIMENTAL RESULTS 47
A. PARTICIPATION IN THE EXPERIMENT 47
B. SCOPE AND CONDUCT OF THE EXPERIMENT 47
C. RESULTS OF THE EXPERIMENT 52
D. INTERPRETATION OF THE RESULTS 54
E. CONCLUSIONS 55

VII. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS 57
A. SUMMARY OF CONTRIBUTIONS 57
B. WEAKNESSES OF MEBUILDER 57
C. FUTURE RESEARCH DIRECTIONS FOR MEBUILDER AND METUTOR 59

APPENDIX A. MEBUILDER SOURCE FILE 63
TAB 1. MEBUILDER MAIN MODULE 64
TAB 2. MEBUILDER CLASS MODULE (MEBuildCLS) 65
TAB 3. MEBUILDER TASK MODULE (MEBuildTSK) 73
TAB 4. MEBUILDER LESSON MODULE (MEBuildLES) 79
TAB 5. MEBUILDER LESSON COMPILER (MEBuildCMP) 83
TAB 6. MEBUILDER LIBRARY MANAGER (MEBuildLIB) 87
TAB 7. METUTOR VERSION 29 SOURCE 92

APPENDIX B. MEBUILDER USER'S MANUAL 97

APPENDIX C. SAMPLE SCRIPT RUN WITH MEBUILDER 117

APPENDIX D. SAMPLE DATA FILES 155
TAB 1. LIBRARY DIRECTORY FILE 156
TAB 2. OBJECT DEFINITION FILE FOR PILOT 158
TAB 3. OBJECT DEFINITION FILE FOR AIRCRAFT 160
TAB 4. TASK DEFINITION FILE FOR PREP_AIRCRAFT 163
TAB 5. LESSON DEFINITION FILE FOR PILOT_TRAINING 168
TAB 6. METUTOR READY FILE FOR PILOT TRAINING 172

APPENDIX E. SCRIPT RUN OF METUTOR ON AN MEBUILDER FILE 187

APPENDIX F. EXPERIMENT CONDUCTED USING MEBUILDER 193
TAB 1. GENERAL INSTRUCTIONS FOR THE EXPERIMENT 194
TAB 2. SUBJECT MATTER FOR THE EXPERIMENT -- SUITE ONE 196
TAB 3. SUBJECT MATTER FOR THE EXPERIMENT -- SUITE TWO 197
TAB 4. SAMPLE RUN OF THE DATA COLLECTION PROGRAM 198
TAB 5. INITIAL DATA FILES FOR SUITE ONE 200
TAB 6. INITIAL DATA FILES FOR SUITE TWO 205
TAB 7. RAW EXPERIMENTAL DATA COLLECTED 210
TAB 8. SELECTED COMMENTS FROM THE PARTICIPANTS 211

LIST OF REFERENCES 213

BIBLIOGRAPHY 217

INITIAL DISTRIBUTION LIST 219

ABSTRACT

Many military applications for intelligent-tutoring systems focus on the training of procedural skills. However, while there have been many successful research efforts into developing tutoring systems for specific applications, the question of developing general-purpose ones is still open. Specifically unsolved is how a lesson-authoring system, a program written to help a novice write computerized lessons, can be made both general purpose and easy to use.

MEBuilder is a prototype lesson-authoring system which employs an object-oriented approach to solving this problem. MEBuilder combines automated object, task, and lesson modeling tools with a library management system to allow teachers to develop simulation-based procedural trainers on nearly any subject. Teachers create reusable objects which have a fixed and well-defined behavior. Then by using the power of means-ends analysis, MEBuilder helps the teacher build entire tasks with these objects in just one step. With these tasks, teachers use MEBuilder's workbook structure to create a lesson containing several exercises. At each step, MEBuilder's automatic error and consistency checking reduces time spent on testing and debugging. MEBuilder's library manager ensures object and task reusability. This thesis explains MEBuilder's design, data structures, and interfaces. It also presents experimental results which support MEBuilder's methods as being more efficient and authoring systems using traditional computer-aided instruction (CAI) techniques.

I. INTRODUCTION

The standard classroom environment contains two primary roles: teachers and students. However, the teachers' role is multidimensional. Teachers must present lessons to the students, monitor their performance, and develop the lesson material. Research efforts into Intelligent Tutoring Systems (ITSs) have produced various computer-based models for these three teacher roles. However, many find the third role, that of developing the lesson material, to be the most difficult challenge for ITS developers.

Developing lesson material is a specific application for an authoring system. Authoring systems are programs that produce other programs in either a programming language or a scripting language for use by another computer program. In addition, authoring systems hide the details of the target language from the user, so the user does not need to know how to program a computer to use it (Heift, 1994, p. 263). There does not yet exist a truly general-purpose authoring system for an ITS. However, there have been several successful research efforts in widening the scope of authoring systems.

This thesis presents MEBuilder, a prototype lesson-authoring system for procedural skills. MEBuilder authors lessons for METutor, an ITS shell written at the Naval Postgraduate School which tutors procedural tasks in virtual-world simulations.

A. THE DESIRE FOR HANDS-ON STYLE TRAINING BY COMPUTER

Military applications emphasize learning by doing (TRADOC, 1991). However, due to a lack of available resources, military training is often conducted in a traditional classroom setting. With respect to procedural skills, classroom instruction is an inadequate substitute for learning by doing. Computer-based simulations provide a better substitute. Simulations provide a reasonable hands-on training environment while not requiring many resources (Psotka, 1987, p. 5). In addition, computer simulators are flexible enough for use in many applications.

Additionally, computers can combine simulations with intelligent-tutoring techniques to provide automated trainers for students. These trainers have many benefits for students. Students can learn at their own pace with minimal direct supervision of a teacher. Trainers give feedback to the students, so the students are freer to make mistakes without risking damage to actual equipment. Simulation environments generally provide "discovery-rich" tools that let students explore the environment unimpeded (Barr, 1982, p. 291), which help make the simulation believable and fosters greater learning.

Unfortunately, these trainers tend not to help the teacher who must prepare the lesson. Simulations are difficult to build and test, but this difficulty is alleviated through tools such as authoring systems. For this application, an authoring system should allow objects in one simulation to be reusable in other simulations. The system should be flexible to allow a teacher to make adjustments easily, but also to check any adjustments to ensure consistency with the rest of the simulation. Finally, the system must save time for the teacher. In order for the authoring system to be effective, it should be based on a modeling technique that works well in simulations and that teachers can readily understand. One technique that appears promising is object-oriented modeling.

B. OBJECT-ORIENTED MODELING AS AN ANSWER TO THE NEED

This thesis employs object-oriented modeling because it provides all the benefits listed above easily and efficiently. Object-oriented modeling is a way of constructing entities in a virtual world so each entity maintains its individual behavior, and interrelations between entities are strictly controlled. An example of using objects in an intelligent-tutoring system application is presented in the aircraft preparation problem given in Appendices C through E. The pilot and the aircraft are the two main objects in the lesson, and the lesson describes specific rules on how the pilot's actions affect the aircraft.

Object-oriented modeling has four beneficial characteristics -- identity, classification, polymorphism, and inheritance (Rumbaugh, 1991, p. 2). Identity means that the data is partitioned into discrete, distinguishable objects. Objects in a virtual world for a flashlight repair problem might include the flashlight, its bulb, and its batteries. Each of these objects would exhibit behavior, and then the three objects share behavior as a whole. Classification

means that similar objects can be grouped together. Polymorphism means that the same operation may behave differently on different objects. Inheritance is a means of establishing a hierarchical relationship among objects, such as the relationship between the abstract object "vehicle" and its refined objects "wheeled vehicle" and "tracked vehicle".

By using identity and classification, objects can easily model the behavior of their real-world counterparts, thus providing the basis for computerized domain knowledge. Polymorphism and inheritance help reduce the size of the data by minimizing repetition -- all the information common to vehicles is present once in the vehicle description, and refined objects only contain the refinements. Finally, using all four to their greatest advantage, one can easily create a wide variety of situations based on modifying one or more objects in a scenario or introducing more objects. Thus it appears that an object-oriented modeling approach would serve well as a basis for a lesson building tool, and with such a model building an accurate representation with a virtual world scenario would be simple and efficient.

C. MEBUILDER -- AN OBJECT-ORIENTED LESSON AUTHORING SYSTEM

MEBuilder is a simple-to-use but powerful lesson-authoring system built using Quintus Prolog. Based on means-ends analysis, MEBuilder was specifically designed to address the needs of virtual-world construction while providing direct access to the existing METutor ITS platform to test and validate lesson material. The primary features of MEBuilder include object-oriented modeling, task modeling, and lesson modeling. It is important to note that METutor is not object-oriented, so these models are new to this platform.

MEBuilder's object-oriented modeling will allow a teacher to construct a simple representation of the make-up and behavior of any object the student must manipulate. In addition, the modeling system allows the teacher to create virtual characters whom the student must interact with. Its task modeling allows the teacher to build the specification for a task that a student must perform and test the object models to ensure that the task is both feasible and correct. Finally, the lesson model provides the framework by which a

teacher may specify a multitude of scenarios and levels of difficulty so a student may learn the basics first and then learn advanced concepts.

D. CONTENTS OF THIS THESIS

1. Chapter II -- Survey of Related Work in Lesson Authoring

This chapter will present other efforts in the field of Intelligent Tutoring Systems and lesson authoring. It will discuss the factors that good lesson-authoring systems must have, and explain why good general-purpose authoring systems are so difficult to build.

2. Chapter III -- Tutoring System Virtual Worlds and Object Modeling Techniques

This chapter will present object modeling as a solution to some of the problems present in Chapter II. It will describe the benefits and pitfalls of object modeling. Finally, it will describe those features that a lesson-authoring system must have to effectively employ object modeling.

3. Chapter IV -- An Introduction to the MEBuilder System

This chapter will present the design and philosophy of the prototype MEBuilder system, emphasizing the specific criteria of lesson-authoring systems that MEBuilder satisfies. The chapter will discuss in abstract terms how MEBuilder represents objects, employs inheritance and other object-oriented design techniques, how it models tasks, and how it constructs a lesson scenario. It will also describe the teacher's interface and how it makes the job easier for the teacher. Finally, it will describe MEBuilder's library management features which help track all the lesson material produced.

4. Chapter V -- Translating an MEBuilder Lesson to an METutor Lesson

This chapter will discuss the relationship between MEBuilder and the underlying tutoring system, METutor, Version 29. In particular, it will present the data structures of an METutor lesson and describe the algorithms MEBuilder's compiler uses for generating them.

5. Chapter VI -- Experimental Results

This chapter will catalog an experiment conducted to test the capabilities of the MEBuilder system. The experiment involved a class of students in artificial intelligence charged to author a given lesson in MEBuilder and a traditional CAI-style system.

6. Chapter VII -- Conclusions and Future Research Directions

This chapter will summarize all of the above, and will describe those areas in which MEBuilder is presently inadequate. It will also present "hooks" into the MEBuilder system left so that integrating some of these advancements will be easier.

7. Appendices

Appendix A contains the header comments to the source code of the MEBuilder system. The entire source is not given because it is several hundred pages long. The complete source is available in a separate technical report. Appendix B contains the text of the User's Manual for MEBuilder, minus the appendices. Appendix C contains excerpts of an MEBuilder session. Appendix D contains example data files produced from the MEBuilder session conducted in Appendix C. Appendix E contains excerpts of an METutor session running the compiled lesson in Appendix D. Appendix F contains the details of the conducted experiment -- including the task given to the students, the raw data collected from the experiment, selected comments from the students, and selected material produced.

II. SURVEY OF RELATED WORK IN LESSON AUTHORING

Many successful research efforts in Intelligent Tutoring Systems provide lesson-authoring features of varying degrees. This is because many researchers recognize that Intelligent Tutoring Systems (ITSs) with a fixed knowledge base and courses of instruction outdate themselves too quickly and cannot handle any special needs of the individual teacher or student. However, there are many differing opinions over what makes a good authoring system. This chapter will present some of the commonly agreed-upon traits. It will then discuss related research efforts and what authoring services they provide. Finally, this chapter will introduce MEBuilder and describe the authoring features it will provide.

A. ELEMENTS OF A GOOD LESSON-AUTHORING SYSTEM

From an architectural point of view, authoring systems have four major components (Elson-Cook, 1994) -- components that construct the domain model, the instructional strategy, the student model, and the communication model. Conversely, the ITS consists three primary components -- the expertise module, the student module, and the tutoring module (Barr, 1982, pp. 229-234). The goal of the authoring system is to provide each of the ITS components with domain-dependent information -- and typically the instructional strategy and expertise module interrelate, as do the communication model and the tutoring module.

The list of properties of good authoring systems is still openly debated, and new applications seem to be producing newer requirements or desires. However, there are recurrent themes in the literature that are consistent with the interrelations among ITS components and authoring system components.

First, the system must contain "theory-rich", not "theory-neutral", tools. (Guralnik, 1994, pp. 235-236) Theory-neutral tools are those that place the burden of creating the learning environment on the teacher. They tend to accent the lesson interface with the

student without providing tools for assisting the teacher in creating consistent instructional strategies. Theory-rich tools have an understanding of the ITS shell to which the system is authoring. This way, they provide means to help the teacher formulate strategies and insure their completeness and consistency.

Second, the authoring system must provide ways for the teacher to encode domain-dependent error feedback to the students. If it was solely left to the ITS platform to model some basic student errors such as misapplication of operations without any domain influence, the chances for a proper diagnosis of an error is slim (Heift, 1994, pp. 263-264). However, at some point, the platform must be able to provide information to the student in words that fit the context of the application. This information, called "evaluative feedback" essential, in order for the student to learn the material being taught rather than simply learning how to solve a computer-based riddle. (Heift, 1994, p. 263) This evaluative feedback is an example of how domain model information is passed parametrically to the instructional model. The best authoring systems make this information flow as transparent as possible.

Third, the authoring system must present a high-level interface. Most teachers will not know how to program. Clearly, the wrong approach is to force teachers to learn to program, and that a better approach is to have the system provide high-level tools that do the programming automatically. (Jones, 1994, p. 300). However, the system cannot be made so simple that the domain of potential lessons it would author is limited (Feifer, 1994, p.198). What the system must do, then, is provide many powerful features in a manner that do not overwhelm the teacher nor limit his/her options.

Finally, for the purposes of serving procedural tasks, the authoring system should be flexible for use in multiple domains. This does not imply that an authoring system built for a specific domain is bad. It certainly is possible that some domain-specific authoring systems could be developed that still provide enough flexibility to be used in other domains. However, if the ITS is purely domain-specific, then for its authoring system must be a completely separate entity that can be used with another ITS.

B. TYPES OF LESSON-AUTHORING SYSTEMS

1. Domain-Specific Authoring Systems

There have been hundreds of successful ITS applications that provide single domain instruction to a student. These applications cover subjects ranging from operating kraft boilers (Woolf, 1987, p. 413), tutoring radar mechanics (Tenney, 1987, p. 59), understanding and writing Chinese characters (Ki, 1994, p. 323), and reading and writing skills (Carlson and Crevoisier, 1994, p. 111). All of the above systems plus many others have the domain knowledge built into the system, so teachers cannot use these ITSs for teaching other subjects.

Moreover, some of these ITS platforms do not provide authoring capabilities. Also, for those that do provide authoring, the authoring system is embedded in the ITS and cannot be used for other domains. Therefore, these systems fail in the fourth criteria.

2. Systems Built Using General-purpose Hypermedia Platforms

General-purpose hypermedia platforms provide excellent environments for teachers to build powerful tutoring applications (Moreland, 1994, p. 748). Several such platforms are commercially available. However, tutoring systems created from these platforms can fall short because the platform does not have built-in tools for producing a true learning environment (Guralnik, 1994, pp. 235).

It falls upon the teacher to create this environment within the context of the platform, which is difficult at best for several reasons. First, such platforms have no knowledge or concept of pedagogical objectives. The teacher must try to create them and hope the platform can convey those objectives within its hypermedia web. Second, they force the teacher to take a well-defined task and map it to sequences of user-interface objects. In effect, the teacher must become a programmer. (Guralnik, 1994, p. 236) Therefore, these systems fail the first criteria we wish to meet.

3. Domain-independent Programmed Tutoring Systems

The ventures into domain-independency began with the development of ITS's that required the teacher to write the lesson in a programming or scripting language. The ITS effectively becomes an interpreter of the teacher's program. Examples include the PIXIE system (Sleeman, 1987, pp. 247-248), which required the teacher to write rules in LISP. PIXIE's authoring system consisted of the text editor used to write the program. Although both emphasize domain independence, such an approach fails the third criteria for evaluation.

C. OTHER ISSUES REGARDING LESSON-AUTHORING

A major concern among lesson-authoring systems is time. By one estimate, it takes an average of 100 man-hours to produce one hour of instruction. This time is the combination of implementing the domain, student, and tutoring models -- it does not even include domain research since it assumes the author is already familiar with the domain. (Murray, 1994) Several other projects emphasized that authoring time per hour of instruction is a concern (Gescei, 1994, p. 15; Jones, 1994, p. 299). Any successful lesson-authoring system must find ways to cut into this implementation time.

Another concern is that any hands-on style training must be believable. Simulations are not a perfect representation of reality since storage space limitations will always force authors to leave out key details. Doing so runs the risk of making the simulation appear canned, which will reduce the effectiveness of the lesson (Feifer, 1994, p. 198). Unfortunately, this is a difficult statistic to measure and is equally a reflection in the choreographic abilities of the author as it is a function of the tools. However, poorly designed tools can hinder the author. The authoring system should allow teachers to visualize the task being built so they can properly evaluate it and make it more realistic.

D. METUTOR -- A MEANS-ENDS BASED INTELLIGENT TUTORING SYSTEM

The first decision for building an authoring system is to find an ITS shell that either does not have one or has an inadequate one. The best ITS shell is one designed for use as a general-purpose ITS capable of handling a wide range of lesson domains. The advantage is that the target representation of the domain material is known in advance. Thus, the lesson-authoring system behaves as both an interface and a translation routine between the lesson as the teacher sees it and the lesson as the ITS sees it. For these reasons, METutor was selected as the ITS shell for an authoring system.

METutor is a pure means-ends based tutoring system developed at the Naval Postgraduate School by Neil C. Rowe (Rowe, 1993, pp. 317-323). METutor is a general-purpose tutoring shell like PIXIE, except that its data representation resembles more of a database than a programming language. Lesson definitions are very simple, and by using the power of means-ends analysis the simplicity still provides a robust platform.

First, METutor only required the teacher to describe the lesson using a minimum of seven predicates (the four standard means-ends predicates -- *recommended*, *precondition*, *addpostcondition*, and *deletepostcondition* -- a random-event template called *randchange*, and the *start_state* and *goal*). METutor also provides seven additional interface-based predicates that allow a teacher to build the lesson in a graphic interface system.

Second, METutor uses Prolog facts rather than LISP notation in describing the rules. Although LISP has a simpler syntax, Prolog facts are easier to read. For example, LISP uses parentheses as the universal grouping symbol for program statements, data lists, etc. Prolog uses parentheses for grouping fact arguments, and square brackets for grouping data lists.

However, METutor lessons are still built as text files that behave like interpreted programs. The teacher uses a text editor to write the lesson, which is loaded with METutor into a Prolog session. Because the means-ends space can be very complex, some logical errors can be difficult to detect, and teacher can become easily bogged down by simple

programming errors. Yet, even if every individual operation is properly specified in METutor, there is no guarantee that METutor will derive the solution that the teacher intended and there is no guarantee that every solution the teacher intended will work in METutor.

Based on the criteria established for good authoring systems and the above description of METutor, the authoring system must provide several specific capabilities to be effective. First, it must establish explicit entities that have well-defined and consistent behaviors. Second, it must encapsulate procedural behavior with sequences that teachers understand, that the system would translate into a set of means-ends rules. Third, it must provide facilities for re-using repeated information so that the teacher can save time. To accomplish this, the authoring system requires an appropriate modeling technique -- and the one chosen for METutor's authoring system is called object modeling. Object modeling is presented in the next chapter.

III. TUTORING-SYSTEM VIRTUAL WORLDS AND OBJECT-MODELING TECHNIQUES

Most of the sample lessons studied using the platform described in the previous chapter plus others (Woolf, et al. 1987) present a virtual world to the student. These virtual worlds contain objects and actors that have both independent and interdependent behaviors, and the student's goal is to satisfy a set of objectives for each object in the virtual world. Normally, virtual-world designers use object-modeling techniques (Rumbaugh, 1991, pp. 57-91), some of which are described below.

A. OBJECT-MODELING TECHNIQUES

1. Generalization

Generalization is the relationship between an object and one or more refined versions of it. In artificial-intelligence terms, generalization is similar to the standard *is_a* or *a_kind_of* relationship between objects. Virtual-world modeling makes extensive use of generalization in order to take advantage of the similarities among similar objects. For example, if all cars have four wheels then one could use that information to save time when describing various types of cars.

2. Aggregation

Aggregation is equivalent to the *part_of* relationship. An aggregate object is treated as a unit, even though it consists of many lesser objects. Further, aggregate objects behave according to a part-whole relationship where the condition of a part of the object affects the whole. Aggregation also describes the *possesses* or *has_a* relationship. Implementation-wise, there is little difference. But conceptually, *has_a* implies more of a temporary ownership. *Part_of* tends to imply a permanent ownership, and the dysfunction or lack of the part renders the whole ineffective.

3. Metadata

Metadata is data that describes other data. Among the uses of metadata are: instantiation, relating a class of objects to a particular instance of that class; summary information, which describes a set of information about an object using a single fact; and null information, which describes whether or not the information is known, applicable, etc.

4. Events and States

Events are things that happen at some time. For example, *Flight 45 departs from Hartford* or *the user has clicked the right mouse button* are events. Some events preceded other event while other events might be completely unrelated -- for example, Flight 45 must depart Hartford before it can arrive in Pittsburgh but neither of these events are relevant to the user clicking the right mouse button.

The state of an object is a set of values that the object holds that affects it overall behavior. States specify the response to input events. For example, if an airplane's engine is unserviceable then the response to an event of *fly this plane to Pittsburgh* will be negative -- whereas a completely operational plane will perform the task.

B. USE OF OBJECT-MODELING TECHNIQUES IN METUTOR LESSONS

METutor makes extensive use of events and states, but not generalization and aggregation. METutor presents the state of the virtual world to the student at each turn, and each operation the student selects constitutes an event. In addition, METutor provides for the definition of random events that are triggered either by the start of the lesson or by a student's action. After the student applies his chosen event, METutor updates the current state, applies all random events, and shows the student the new state.

However, one shortfall in METutor's modeling of states is that there is no precise relationship among mutually exclusive members of a state. For example, flashlight's may either be on or off. Naturally, any event that causes the flashlight to become on must delete the previous fact that the flashlight is off. Unfortunately, there is no direct way to specify that "on" and "off" are opposite states in METutor. This means that a teacher may forget

to specify the "delete off" parameter in this event and METutor will likely not detect the error.

Because METutor uses a simple Prolog-fact structure for the state, its lessons have a limited sense of the objects contained in the virtual world. Prolog facts use symbols as arguments to predicates, and teachers assume that predicates with like arguments constitute a combined state of an object. However, there is no direct way to model aggregation other than employing a programming convention that aggregate objects refer to its components via the possessive. In other words, let's assume that a teacher writes as a component to an METutor fact, **disassemble(the,flashlight's,top)**. METutor does not recognize that **flashlight** is an aggregate object nor does it understand that the **top** is part of that aggregate. The flashlight's top is simply another argument set. The drawback is that the teacher must directly specify and manage these component relationships, a difficult task to do error-free.

Similarly, METutor has no means of recognizing that two objects are instantiations of the same object type. For example, a teacher is writing a firefighting problem where the fire team leader has two subordinate teams at his disposal, called red team and blue team. Both the red team and blue team objects are identical instances of a class of firefighting teams. However, in METutor, there is no facility to describe both objects using a common set of events and states. Instead, the teacher must repeat all the rules for both teams and directly include mutually exclusion clauses to prevent the two teams from performing the same action. The potential for errors is great. Teachers could very easily fail to specify all the rules when preparing the copy for the second team, or accidentally insert **blue** instead or **red** somewhere in that second set. Additionally, in a larger system with more than two like objects -- such as a system administrator lesson where a system has hundreds of users -- this process is both tedious and unnecessary.

Finally, METutor lessons don't have a built in state diagram, so there is no direct temporal relationship of states visible to the teacher. This means that if the teacher is not careful, he could produce a lesson that has unintended solutions. An example of this was

a case where a student wrote a lesson for a fifteen-step procedure that had a possible solution of no actions[1].

C. OTHER BENEFITS TO USING OBJECT MODELING

1. Object Re-Use and Customization

A well-defined set of objects could serve as the basis for an entire set of virtual-world based lessons. For example, most naval firefighting equipment is similar among the various classes of ships. Firefighting teams are also similar in make-up and behavior. So by defining the make-up and behavior of a general class of firefighting teams and equipment, a teacher could develop a series of lessons in firefighting among a large variety of ships without having to repeat information. Since virtual-world descriptions can become quite large and complex, this is a definite time-saver. Reusable objects also provide the teacher with the comfort of consistent behavior, so the teacher can confidently use the object in any lesson and count on its behavior.

Also, subtle differences in the make or model of two objects could be sufficient to desire to customize the model. For example, the Army's AN/GRC-142 Radioteletype System has seven different models -- GRC-142 through GRC-142F. However, the differences among models are not subtle -- some are wholesale component changes, and some Army units had special versions of one of the models as a result of a component's experimental fielding. Teachers in these units will appreciate an ability to customize objects for local use.

2. Encapsulation

Encapsulation, or information hiding, protects objects from unintended change. It will allow a teacher to substitute objects in a lesson while preserving the behavior of the

---

1. This actually occurred during tests on the first prototypical authoring system for METutor. A student who apparently misunderstood the meaning of states coded identical start and goal states of the lesson. Even though the lesson as written worked properly, the quickest solution was to do nothing.

other objects in the scenario. For example, a posed problem for a ship refueling problem might involve a cruiser. Then a second problem might involve a battleship. The different ships have independent behaviors, but their relation to the fueling ships does not change.

D. PITFALLS TO USING OBJECT-MODELING TECHNIQUES

1. Generalization is Not Equivalent to Subclassing

Object-modeling is not an easy task, and even in the well-defined realm of programming languages there still exists heavy debate regarding what constitutes a valid class and object hierarchy (Lalonde and Pugh, 1991, pp. 57-62). The concepts of generalization and subclasses are not equivalent, meaning that an *a_kind_of* relationship between two object types does not necessarily constitute a proper class hierarchy. Lalonde defines subclassing as "an implementation mechanism for sharing code and representation," whereas generalization is a "....specialization relationship, i.e. it describes one kind of object as a special case of another." Figure 1 describes a relationship among data structures that clearly shows the difference among the two. Although binary trees are clearly a special type of directed graph, the most efficient data structure to implement the binary tree is a node with two son pointers. Directed graphs, meanwhile, are best implemented using an adjacency matrix. Since such relationships are not always clear to computer specialists, the potential to confuse a relative novice such as a teacher is high.

This problem arises in modeling objects in virtual worlds. For example, an electric-powered car, such as a golf cart, is *a_kind_of* car. Clearly one cannot derive the behavior of a golf cart from car because the two have completely different engines. A teacher desiring to build a golf cart object must survey the existing subclass hierarchy to insert properly entities such as golf cart. Unfortunately, there is very little any object-oriented environment can do to detect and correct errors. It is incumbent on the teacher to recognize and use the best hierarchy possible.

2. Increased Complexity for Simple Lessons

Lessons that are very simple -- those with few objects and a small number of events or states -- should remain simple to write. Adding object-modeling inherently increases the complexity by requiring the teacher to define the model. Therefore, defining the model for simple objects must not serve as a deterrent for the teacher.

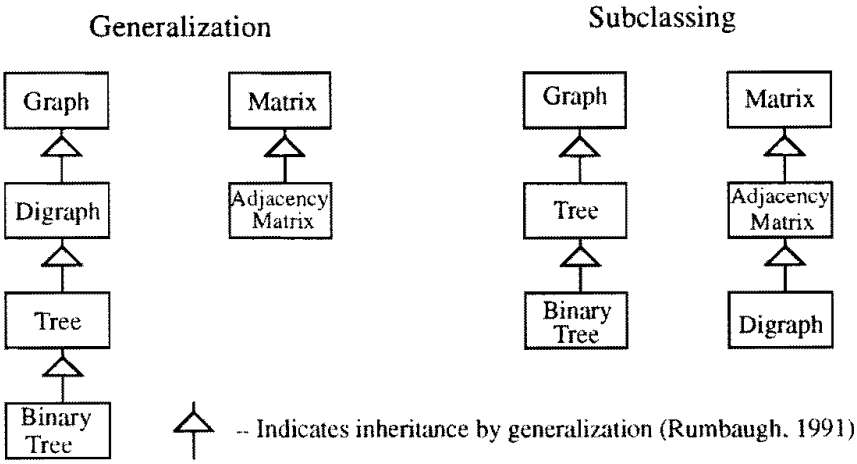


Figure 1: An Example of Differences in Generalization and Subclassing Hierarchies

E. TOOLS NEEDED TO ADD OBJECT MODELING TO AN AUTHORING SYSTEM

Clearly, in order for an authoring system to reap the benefits of object modeling without becoming an excessive burden for the teacher, it requires specific object-building tools. These tools must serve the purposes of providing a simple framework for defining the generalization hierarchy; for defining aggregations, events, and states; for specifying inter-object relationships; and for providing immediate error-checking to the teacher. In

addition, the tools should guarantee to the teacher that the lesson is complete and accurate before use by a student. But most importantly, these tools must abstract the teacher from the low-level representation of the objects and speak to the teacher in a language that he can easily understand.

1. Object-Modeling Interface Concept -- the Adventure Game Model

Many modern-day three-dimensional adventure games present a virtual-world to the game player as if the virtual-world were a stage and the student is acting in a role (Example: Sierra, 1993). This conceptual model is extremely easy for a teacher to understand, yet still provides an adequate framework for building all the other tools necessary to construct the virtual world. Therefore, one good way to assist the teacher is describing things in stage terminology -- a cast, a set of props, and settings.

The cast is an easy model to demonstrate how the tools can provide yet hide generalization and aggregation to the teacher. Typically, a cast member plays a role, which is a subclass of a character. This role generally exhibits a specific set of behaviors -- actions that he must perform to respond to external stimuli. Roles can be generalized and can maintain or possess props -- plumbers and electricians are *a_kind_of* handyman, for example, and handymen possess tool kits. Once a teacher defines a particular role, he instantiates it by inserting it into a particular virtual world. In addition, the teacher could specify that the student is to play a particular role and perform some task in the scene. Similarly, stage terminology will help simplify the process of defining props and settings for the teacher.

2. Object-Modeling Tools

The above paragraph describes how a platform can make generalization and aggregation easier to model. Events and states also have properties that an object-modeling platform could exploit to save time for the teacher.

a. Mutual Exclusion of States

Most potential states that an object can attain are mutually exclusive of other states. Previously, this thesis presented an example of how a flashlight could not be both on and off -- it must be one or the other. Similarly, a fire is either raging or out; a street light is either green, amber, or red; and the chloride level of the water in a ship's boiler can have precisely one value in the range of 0.0 to 1.5 parts per million. Especially regarding qualitative values, the tools use these teacher-defined mutually exclusions and ensure all operations involving the object maintain this property.

b. Summarizing Object States

A car's engine could contribute hundreds of items to the state. In many applications, not all of these items are relevant or interesting to the student. For example, if the fuel injection system is working, the student probably only needs to know that it is working -- not the ten or twenty different state members that make up the fact that it is working. Providing tools to describe summaries of large groups of data will not only provide greater expressive power to the lesson but simplify the lesson building process for the teacher.

c. Modeling Unknown Information

A very common property among diagnostic lessons is that the student does not know all the information that is true in a state. However, any lesson could have a need to model what a student knows or should know. There are several specific properties which make knowledge easy to model and thus easy for tools to implement. Moore developed a full first-order logic theory concerning modeling what a student knows and what actions a student takes based on what he knows. In his theory, Moore describes things known and unknown as additional members of the state influencing the end result of the theorem being proven (Davis, 1990, pp. 390-391).

Many props are consistent regarding what an actor would know or not know about the object. For example, a student would not know that the batteries inside a

flashlight are dead just by looking at the flashlight. Instead, the student would need to engage in a series of steps to investigate the dysfunction of the flashlight and thereby soon discover the problem with the batteries. The status of the batteries is something that would normally not be known by a student, so the object model of a flashlight could contain an additional flag marking the battery status as hidden.

d. Modeling Objectives

In order for a teacher to build the lesson, the teacher must be able to describe what the objectives of the lesson are so the ITS platform may identify when the lesson is over and identify points where the student is failing to make progress. Additionally, the teacher must identify objectives for all the other actors in the scene so that their behavior is consistent and, to a degree, predictable.

e. Modeling Operations and Sequences of Operations (Tasks)

To provide a useful method of describing the actions a student may take, teachers must combine a student's potential objectives with a description of atomic operations that cause the change of state of an object. Together, the platform should help the teacher describe all possible sequences of atomic operations, or tasks, that achieve those objectives.

Atomic operations, operations that a teacher decides he cannot or will not break down into subactions, consist of several components. First, operations have a direct object -- the specific object being manipulated. Second, operations have a list of indirect objects -- those other objects required for performing the operation. Third, operations have an intended effect -- the specific change of state that the direct object will attain as a result. Finally, operations have preconditions -- the required states that all direct object and indirect objects must be in to apply the operation. With these atomic operations, the tool has enough information to make a partial assessment of what actions are necessary to achieve a given objective from a given staring state.

F. SUMMARY

This chapter has presented a brief study of authoring systems for ITS platforms and described how object-modeling techniques could enhance their functionality. In addition, this chapter discussed several object-modeling issues and described how an authoring system could use object modeling to help teachers build lessons. The next chapter will describe MEBuilder, a prototype authoring system for METutor, which employs object-modeling techniques.

IV. AN INTRODUCTION TO THE MEBUILDER SYSTEM

MEBuilder is a lesson-authoring system written entirely in Quintus Prolog, taking advantage of several Quintus Prolog library modules. It presently uses only text for input and output, but its design lends itself for use in a menu-driven windowing environment.

A. MEBUILDER'S TOP-LEVEL DESIGN AND PHILOSOPHY

MEBuilder consists of one main program module and five primary submodules. Appendix A contains the header comments for these modules. Its overall design parallels that of the Ada Programming Support Environment (DoD, 1983). Not only does MEBuilder provide the editing capability for building lesson material, but also provides library services that cross-checks the lesson material for consistency. Figure 2 is a diagram showing the relationships among MEBuilder's primary modules, its primary data stores, and the external tutoring system.

1. MEBuilder Lesson Material -- The Three-Layered Design

The three modules across the middle of Figure 2 indicate the three modules corresponding to MEBuilder's three-layered lesson design -- the three layers being classes, tasks, and lessons. The purpose of this design is to maximize code re-use and reduce authoring time. The intent is that the final version of MEBuilder would come with whole libraries of classes and tasks, and the focus of the teacher falls solely on the lesson layer.

a. The Class Layer

The class layer is where the basic object descriptions lie. It corresponds to the class data structures in an object-oriented programming language. The class layer manages the primitive attributes and values for all MEBuilder objects. Classes are abstract entities that are instantiated during lesson construction.

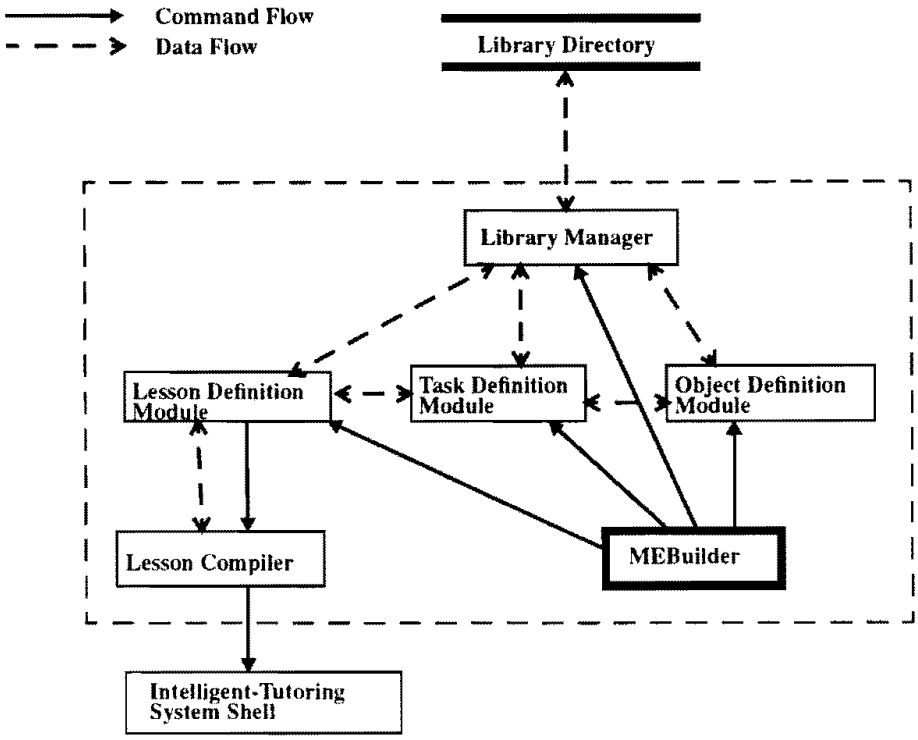


Figure 2: MEBuilder's Architecture

b. The Task Layer

The task layer is where the primitive sequences of operations lie. It corresponds to methods in an object-oriented programming language. A task is an encapsulation of a single initial condition, a single goal, and a method to achieve the goal without external stimuli. The method consists of the defined primitives from the class layer. Tasks are also abstract entities.

c. The Lesson Layer

The lesson layer is a workbook with a collection of concrete problems for the student to solve. The lesson consists of named instances of classes and relevant tasks

to perform, and each problem provides a particular initial setting and objectives for the student to achieve. Well defined classes and tasks make this very simple to accomplish.

The lesson layer also provides access to the underlying METutor system. This access allows quicker testing and debugging of the lesson material.

2. MEBuilder's Bottom-Up Approach to Authoring

MEBuilder uses the three-layered design to enforce a bottom-up approach to lesson design. Thus, the user must design the objects first, followed by the tasks, and finally the lesson itself. Appendix B contains the main text of the MEBuilder User's Manual detailing the process. Appendix C contains an annotated script run demonstrating MEBuilder's main commands.

The bottom-up approach has several advantages. First, MEBuilder performs consistency checks at each step to ensure that changes in a lower layer do not adversely impact data in a higher layer. Second, MEBuilder can use the lower-level information in order to save typing. For example, when building a task, MEBuilder provides the user with menus containing all the appropriate information from the class layer. The user then only selects from menu items rather than having to type the information -- which he would still have to insert into the class definitions later. Third, MEBuilder can use means-ends analysis to assist the teacher in building tasks and lessons. This is not possible without a completed set of class definitions.

There are also disadvantages to the bottom-up approach. First, it is difficult for the teacher to visualize the lesson as he builds it. Clearly, if the teacher has a complete library of objects and tasks, he will spend less time in the lower two layers and minimize this effect. But the worst case scenario will likely be the norm.

Second, the bottom-up approach is vulnerable to modular interface problems. Tasks that individually work might not combine well in a lesson. For example, two task definitions when combined into the same lesson may cancel each other's effects or become mutually exclusive. MEBuilder currently only has limited capabilities for detecting

potential interface problems. Therefore, users must exercise caution when building complex scenarios. Also, all tasks defined for a lesson should be designed as disjointly as possible -- meaning the one operation should only appear in one task.

B. MEBUILDER MAIN MODULE -- "MEBuilder"

The main module performs several key functions. It provides MEBuilder's command loops, its help facility, its autosave facility, and compilation data for Quintus Prolog.

MEBuilder has four command loops -- main, task, lesson, and problem. The main command loop is self-explanatory. The other three are special loops invoked from main which have specific functionality. The relationships among the loops and MEBuilder's three layers is diagramed in Figure 3.

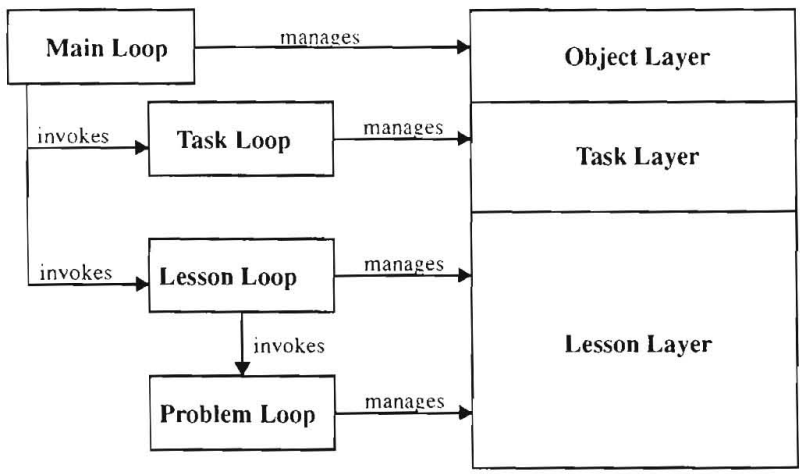


Figure 3: Relationships of Command Loops to MEBuilder's Three-Layer Lesson Design

MEBuilder provides a very simple help system, future versions may include a more context-sensitive facility. The user may enter the help system from any of the four loops and query for information on any command or on various topics.

MEBuilder's autosave facility is a protection mechanism for the teacher. After each tenth command, the entire MEBuilder database is save to an autosave file, called "autosave.meb". The purpose of the autosave is to allow the user a chance to recover to a previous program state if for some reason MEBuilder aborts without warning. A special "restore" command exists in the main loop which restores the saved environment. Currently, the autosave facility parameters are fixed -- the user cannot set the number of turns nor change the name of the target autosave file.

MEBuilder is designed to be run as a stand-alone system. The main module contains all the data necessary for the Quintus Prolog compiler to create a separate executable file from MEBuilder.

C. MEBUILDER'S LIBRARY MODULE -- "MEBuildLIB"

MEBuildLIB's purpose is to save time for the teacher by untying his hands of file management. In MEBuilder, each class, task, lesson, and compiled lesson gets its own file. This makes it easier for the teacher to access them at will. However, there is a clear and defined dependency among the above entities. For example, a class is dependent on its parent class, and if that class is in the database without its parent then the inherited information is not available. Hence, when a teacher requests that a particular entity be loaded, all other entities that the entity depends on are also automatically loaded. Entity dependencies are defined as follows:

- A class is dependent on its parent.
- A class is dependent on the class of each of its components.
- A task is dependent on each class it is built from.
- A lesson is dependent on each class and task it is built from.

To accomplish this, MEBuildLIB creates and manages a special library subdirectory in the user's working directory. This subdirectory holds all the files for the various objects and holds one special directory-information file maintained during the session. The

information file is a Prolog fact file containing entries for each entity -- the entity name, type, file when stored, date of last save, and dependencies.

The date of last save is also important. MEBuildLIB attempts to catch potential errors by checking these dates. If a class has been updated after a task that uses it was last saved, the change to the class might have induced an error in the task. As a rule, changes in a class invalidate all tasks and lessons that use it (compiled lessons based on the task are still OK).

D. MEBUILDER'S CLASS DEFINITION MODULE -- "MEBuildCLS"

MEBuildCLS is a simple data structure manager which serves two important functions. Its primary and most visible function is to build class definitions. Its second function is to provide the other modules with the class information they need in order to perform their functions. It is in this module that all object modeling takes place.

1. The Class Data Structure

a. class_def(<class>, <parent class>)

Currently, MEBuilder only handles single inheritance, and the intent is for the inheritance to model the *a_kind_of* relationship only. MEBuilder recognizes two global classes of which all classes must descend from -- *prop* and *character*. The significance of prop versus character is critical for tasks. There is one *class_def* field per class.

b. component(<class>, <component class>, <component name>, <tense>)

Component multifacts encompass both the *a_part_of* relationship among props and the *has_a* relationship among characters. There is no restriction in MEBuilder regarding what class can serve as a component of what class except that the user must adhere to *part-kind* inheritance and circular inheritance is not allowed. The component name primarily helps distinguish among like components of the same class -- such as unique names for the four wheels of a car. If there is only one component of a given type, the name should equal the class. The tense argument helps METutor print out the correct verb forms for the components whose name does not follow the "ends in s" rule.

c. property_set(<class>, <property set name>, <domain>, <hideable>)

Property_set multifacts describe the various attributes and values that the class can take on. Collectively, the active values of the property sets constitute the object's total state. The members of the domain are mutually exclusive. For example, a streetlight might have two property sets -- "color" with domain "red", "amber", and "green"; and "persistence" with domain "flashing" and "not flashing". Therefore, a light's state could be red and not flashing, or it could be amber and flashing. Currently, domains are limited to being one of a list of qualitative values. Future implementations may include the ability to specify ranges of numbers.

In addition, a property set may be declared as having a possibly unknown or hidden value. For example, a flashlight battery's charge level might not be known be direct inspection -- instead a test meter would be needed. The hideable argument allows MEBuilder to create an extra property set which contains the values "<set> is known" and "<set> is unknown". This information is available for use during task construction and lesson construction. In addition, if a property set is declared as hideable, the teacher may define operations whose purpose is to make the value of the set known to the student.

d. relation(<class>, <relation name>, <definition>)

Relations, or "summary facts" allow the teacher to describe a substate of the object in a single term. A good example of this is with a flashlight. If the flashlight's case is closed, top is assembled, batteries are working, and bulb is working; then the flashlight is working. When METutor prints out the state, the four members of the definition will be replaced with the single phrase "flashlight is working."

e. operation(<class>, <indirect objects>, <operation name>, <intended effect>, <preconditions>, <side effects>).

Operation facts represent the primitive operations that can be performed on an object. By "primitive", this refers to an action that requires one turn in METutor to complete. The indirect object list is a list of all other objects to be present for this operation

to be available. The indirect object can have multiples, including another <class>. The operation's name is an imperative sentence -- a verb phrase followed by a direct object followed by a sequence of prepositional phrases if needed. The direct object must either the <class> or a valid component name of <class>.

The final three arguments describe the behavious resulting from this operation. The intended effect is the one change of state that is the primary reason why the student would perform this action. For example, the student would choose "disassemble the flashlight's case" for the intended effect of "flashlight's case is open". There may be other changes of state among the class or the indirect objects. These are called side effects. The precondition list is a list that describes what state the <class> and <indirect objects> must be in before the operation may be used.

f. daemon(<class>, <daemon name>, <triggering condition>, <advancement criterion>, <advancement form>, <activation message>)

Daemon facts are changes of state that occur as a result of internal, not external stimuli. Usually, they correspond to a sequence or series of state changes which might culminate in some (possibly disastrous) event at the end. No operation is performed to effect the changes induced here, instead the change occurs whenever the triggering condition is true for the object. The activation message is given to the student whenever the triggering condition becomes true. The advancement form describes how often the change in state occurs, either as a probability of change or as the number of turns between changes.

There are three types of daemons. The progressive daemon causes an object to take on the first value of some property set, and advances until the last is reached or the triggering condition is removed. A example of this is the hunger of a person. At the beginning, the person may be full -- but later he progresses through peckish, hungry, starving, and finally dysfunctional or dead. The looping daemon loops through a property set. The streetlight is a perfect example -- it loops among green, amber, then red, then back

to green. The updating daemon invokes an operation. Currently, these are not implemented -- but they are intended for use in updating readings on a meter or other continuous functions.

2. Information Cached to the Other Modules

MEBuildCLS does not send the class' raw data to the other modules for processing. Rather, MEBuildCLS will receive a list of instantiations from the other modules and will return instantiated facts. For example, a lesson has a "John Smith" who is a pilot. The pilot object has a property set of "pilot is cleared for takeoff" and "pilot is not cleared for takeoff". MEBuildCLS will provide an instantiated set of "John Smith is cleared for takeoff," etc.

MEBuildCLS only sends property set data and operation data to the task module MEBuildTSK and the lesson module MEBuildLES. However, all class information is instantiated and sent to the lesson compiler MEBuildCMP.

E. MEBUILDER'S TASK DEFINITION MODULE -- "MEBuildTSK"

MEBuildTSK is by far the largest and most complex module in the system. It serves the purpose of developing procedures made up of the primitive operations of its member classes. However, its underlying purpose is simply to establish relationships among the primitive operations within specific contexts that the operations themselves do not convey.

The name "task" could be misleading. When the teacher builds a task, he describes the entire task in terms of a known starting point and a known goal. The task that is produced is the full task. However, during an METutor lesson it is often that the student may find himself in a situation that puts him in the middle of the task. Here, using the power of means-ends analysis, the student can still complete the task as built.

The key to successfully building a task is providing all possible solutions to the student. MEBuildTSK uses means-ends analysis to help find alternate solutions and solutions where the student may select a different order of operations that will still achieve

the goal. However, it is far better to keep the individual task as small as possible in order to ensure consistency when put together with other tasks in a lesson.

The task's data structure is made up of several components -- basic data components, the procedure graph, and the guaranteed state structure. Not all elements of the data structure reside in secondary storage -- some as cached by MEBuildTSK or from other modules during the session and disappear upon completion of the session. Temporary facts, however, are autosaved.

1. The Task's Basic Data Components

The following is in addition to the temporary information cached from MEBuildCLS.

a. task(<task name>, <actor>, <other objects>)

The task fact indicates those objects involved. All tasks require an actor, which is an object descended from the character class. The other objects may be of any class.

b. initial conditions(<object>, <state>) and objectives(<object>, <state>)

These are self-explanatory in nature, however the respective states are not absolute. The state contains one entry for each property set, but the entry may be a "don't care" value. "Don't cares" help avoid the inclusion of unnecessary operations in the task and provide greater flexibility for the student.

2. The Task's Procedure Graph

Although many tasks are described as a linear sequence of steps, few tasks are truly linear (Sacerdoti, 1990, pp. 162). Many have multiple solutions based on the fact that some operations can be done in different orders. Rules such as the preconditions embedded in the objects' operations help define which operations must precede others. However, those preconditions effectively describe the behaviour of the object in a vacuum. In the context of a particular task with a specific goal to achieve, new rules are required.

Therefore, a structure is needed that describes the temporal relationships among the objects in the task.

MEBuilder's procedure graph is based on Sacerdoti's Procedure Net (Sacerdoti, 1990, pp. 163-168) and Homem de Mello's and Sanderson's assembly state graph (Homem de Mello, 1991, pp. 229-231). The procedure graph is built based on a first attempt solution to the problem posed by the initial conditions and the objectives. MEBuildTSK then assumes that its solution is the only solution. The facilities provided by MEBuildTSK then allow the teacher to identify alternate solutions, during which MEBuildTSK checks to ensure they are indeed valid solutions.

a. Procedure Graph Structure -- stages and actions

A procedure graph is an extended directed graph where the node is called a stage and the transition is called an *action*. A sample procedure graph is in Figure 4. Actions contain the operation to be performed and the additional preconditions and side effects involved.

Stages are conceptually more complex, as they enforce the following rules regarding the graph structure. First, the graph has one start state and one done state corresponding to the initial conditions and objectives being true. Second, a stage which has one transition out indicates that there is precisely one solution to achieving the next stage. Third, a stage which has more than one transition out indicates multiple solutions in two forms -- called and-splits and or-splits. An and-split indicates that the transitions out of the stage correspond to subprocedures that can be done in parallel. This means that order among the actions is unimportant so long as actions within a subprocedure are done in order. An or-split indicates multiple subprocedures that achieve the subgoal, and the student only must perform one of the subprocedures. Fourth, all splits have a corresponding join stage (shown in Figure 4 as the shaded stage marked with a "J") .

Split-join pairs are strictly nested. Therefore, all splits are joined by the time the done stage is reached. Join stages always have a single null, or lambda, transition

out. The extra join stage is required because a split stage, such as Q1 in Figure 4, can only have one split. On the other hand, Q5 could close multiple splits. This could only be accomplished through the use of a sequence of nested joins, each connected by a lambda transition.

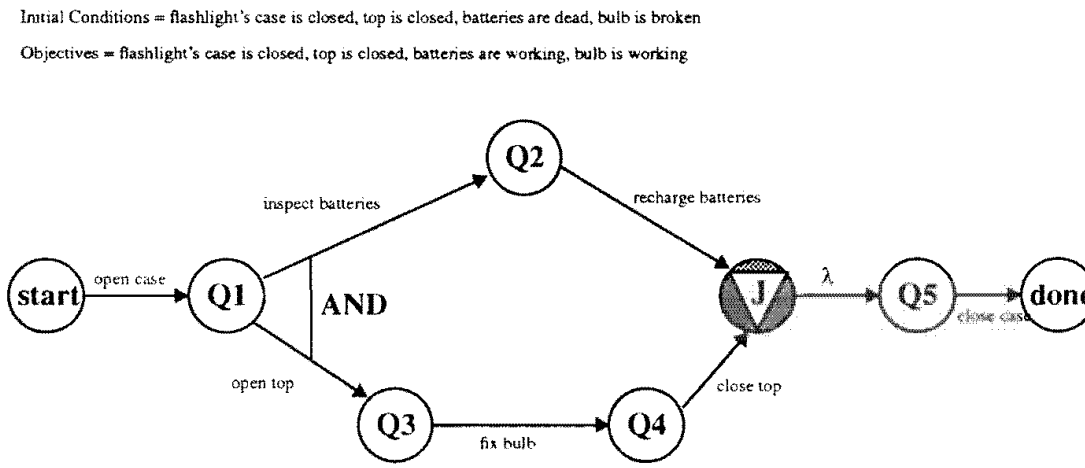


Figure 4: Example Procedure Graph

Procedure graphs may also have unordered actions. These are actions which are required to be performed at some point in the task, but have a very loose temporal relationship with the other actions in the graph. For example, a given device may need to be tested before use during a task. However, one might not care when or where the device is tested, so long as the preconditions for the testing are met and the testing is done before it is used. These are similar to an and-split in concept, however they bypass the strict nesting.

b. Options for Manipulating Procedure Graphs

As stated earlier, MEBuildTSK starts with a single solution and assumes it is the only solution -- so the student must follow the one solution in order. There are many ways in which a teacher can provide, or reduce, the number of solutions available. The fundamental concept MEBuildTSK uses is dependency of the primirive operations. An operation X is dependent on another operation Y if and only if X's preconditions are not disjoint with Y's postconditions. After the teacher performs any of the below, MEBuildTSK invokes means-ends analysis to test the resulting procedure graph.

A teacher may request MEBuildTSK to look for subprocedures that can be parallelized. These are found by examining adjacent actions and seeing if the second is dependent on the first. If such a pair is found, then MEBuildTSK looks for the nearest operation of which both are dependent, then follows the dependencies to identify two or more possible subprocedures. This process produces and-splits in the procedure graph. A teacher may also combine subprocedures together.

A teacher may ask MEBuildTSK to declare an action unordered or ordered. If it is declared unordered, it is marked as such when the user asks to see the procedure structure. If an action is declared ordered, it is placed directly in front of the action that depends on it.

A teacher may move individual actions around within the bounds of the dependency relationship. He may reverse two actions, move an action intoa subprocedure, or move it out of a subprocedure.

Finally, a teacher may also modify the preconditions and side effects of an individual action. This action may lead to MEBuildTSK recalculating the solution for a task or the subprocedure the action is in. If the graph no longer represents a valid solution, MEBuildTSK will scrap the graph and start over.

3. The Task's Guaranteed State Structure

The guaranteed state database parallels the procedure graph and provides information to the teacher about what state corresponds to the completion of some given step at any point in time. It also helps identify particular contexts in which an operation will behave differently than as primitively defined. The latter point is especially true when a primirive operation is specified more than once in a given task or is used in multiple tasks within the same lesson. For the teacher, this structure is useful primarily for information purposes. He may request to see the anticipated state after a given action is completed.

The guaranteed state is a single state for all objects in the task. There is one entry for all property sets among the objects. However, each entry is either a single property which is absolutely true, or is a list of properties in the set which may be true based on probabilities or or-splits.

F. MEBUILDER'S LESSON DEFINITION MODULE -- "MEBuildLES"

MEBuildLES, by contrast, is the smallest module. MEBuilder lessons are merely collections of individual problems which the student can try. Every lesson must have at least one problem, otherwise the lesson is meaningless. MEBuildLES's purpose is to provide command interfaces to the lesson data structure and to provide access to the lesson compiler in MEBuildCMP and the underlying METutor system.

The lesson definition module provides both the lesson loop and the problem loop, as shown in Figure 3. Each manipulate different portions of the MEBuilder lesson.

1. The MEBuilder Lesson Data Structure

MEBuilder lessons have a very simple data structure. The *lesson* fact contains the names of the cast members by name and type (character class), the props by name and type, and the tasks involved in the lesson. The *lesson_intro* fact contains text that appears to the student when the lesson is begun.

2. The MEBuilder Problem Data Structure

MEBuilder problems are numbered in order starting at one, and each problem has its own of the following data items. The *problem* fact contains the name of the problem and those cast members and props that are to be left out in the problem. The *problem_intro* serves the same purpose as the lesson_intro. Each problem has *initial_setting* and *objectives* facts for each cast member and prop, which correspond very closely to the task's *initial_condition* and *objectives* facts.

Finally, though not implemented, hooks have been placed in MEBuildLES where a teacher will be allowed to override some of the probabilities and side effects among the various tasks. These overrides will allow a teacher to increase the level of difficulty of some problems by increasing the probabilities of some bad effects, or make a problem easier by eliminating the possibility of those bad effects.

G. MEBUILDER'S LESSON COMPILER -- "MEBuildCMP"

The compiler takes a lesson which its associated tasks and objects and produces an METutor-ready lesson. The compiler only works for METutor versions 29 and beyond. The compiler only provides a single command to the user -- that of "compile lesson" in the lesson loop. The overall compilation process is simple. First, final integrity checks are performed on the entire class hierarchy for all classes used, followed by the tasks, and finally the lesson itself. Then, each METutor fact is individually constructed. Some METutor facts are required to be cached in a particular order, so sorting routines are invoked as needed. The specific content of the METutor database and how they are derived from compilation is given in the next chapter.

V. TRANSLATING AN MEBUILDER LESSON TO AN METUTOR LESSON

MEBuilder and METutor shared an evolution over the course of this project. As features for one were added, so too was the ability of the other to use it. This chapter will present METutor version 29, emphasizing the key data structures and philosophy changes from METutor version 27, the last active version before this project. It will then present the methodology behind how MEBuildCMP takes lesson material in object-oriented form and produces an METutor-usable lesson for the student.

Appendix D contains sample data files produced from MEBuilder sessions. Tab 1 is a sample library-directory file showing the library entries for the pilot lesson constructued in Appendix C. Tabs 2 through 5 of Appendix D contain sample MEBuilder object, task, and lesson files. Tab 6 is the lesson file translated to METutor form. Appendix E contains an excerpt of an METutor session running this lesson.

A. HIGH-LEVEL DESCRIPTION OF THE DESIGN CHANGES IN METUTOR

1. Workbook Structure Based on MEBuilder's Lesson Structure

METutor's original interface was a simple one-level interface, and METutor lessons were built with only one problem. When the student began METutor, it presented the single problem immediately and exited once the student completed the problem. In the new METutor, the student runs a main command loop which affords him options. When the student selects a problem to do, then a second loop is engaged which runs the problem. A student can exit back to the main loop at any time and retry a problem if he so desires.

In order to ensure backward compatability with lessons written before the workbook format, METutor inspects loaded lesson files for structure. If the lesson file does not contain a workbook structure, then one is built for it.

2. Agents -- Based on MEBuilder Characters

An agent in METutor is a direct manifestation of the character object in MEBuilder. The goals of the tasks involving character objects become a "resting state" for the corresponding agent. Each agent operates using his own set of means-ends rules based on those same tasks. Then as the student takes his turn, all the other characters in the problem take their turn via this agent structure. The student always goes first in a given turn, and the agents follow in arbitrary order.

3. METutor's Macro-Expansion Language

The most visible change to METutor lessons is that they are no longer a set of immediately usable Prolog facts. Rather, they are most macros. It is in the use of macros that MEBuilder's object-oriented philosophies manifest themselves. The introduction of the macro form was based on lessons which had multiple objects of the same type and potentially rules in METutor which changes from problem to problem. The macro form that all METutor rules use follows this form:

<rule-name>_t(<agent>, <quantified vars>, <macro arg1>, ... , <macro argn>)

The macro-indicating entry "_t" is stripped off and the quantified variables are replaced with concrete ones based on the cast and props in the problem. For example, let's say that a problem has two flashlights and that the **recommended** rule dictates that to achieve "flashlight is on" one must "turn on the flashlight". Then, the macro form could be described as follows. The quantified vars would be "for each flashlight x". The first macro arg, corresponding to the target state, would be "x is on" and the second macro arg would be "turn on the x". If we named the two flashlight's in the problem "red flashlight" and "silver flashlight", then the macro expansion would insert those two names for the "x" in the above phrases.

The agent argument can also be an agent domain. For example, if there are five agents all of the same type, then the object type name fills the agent argument. Macro expansion would then produce five sets of rules -- one for each agent.

If there is only one of a given object, then the appropriate macro arguments are pre-expanded in MEBuilder's compilation process. This is because macro expansion only gains speed and space when there are multiples in the object domain. Nothing is gained in expansion for singular instances. Therefore, the actual use of macro expansion in the average lesson is likely to be small.

4. Backward Compatability of Lesson Material

In order to allow backward compatability, METutor version 29 wraps a general workbook shell around the lesson database. Also, the facts are converted to macro language format. This backward compatability is only good for lessons written for the text-based versions of METutor versions 1 though 27. Lesson material for MEGraph versions 1 through 27 will still work, however the graphics facts are ignored.

B. CONCEPT OF THE TRANSLATION PROCESS

Much of the translation process is simply copying data from the MEBuilder lesson layer to the METutor file. Examples of this include the introductory text for the lesson and the problems, cast and prop lists, initial settings, goals, and identification of singular and plural nouns. However, the means-ends rules -- consisting of the recommended, precondition, addpostcondition, and deletepostcondition facts along with the random event mechanism called randchange -- are more complex. All five of these are generated based on the usage of the primitive operations in the tasks loaded in the lesson. Some of these rules must also be sorted so that the higher priority operations are accessed first. Some of these rules are also agent-specific, meaning that they apply only to certain characters in the lesson.

The translation process goes as follows. First, an integrity check is performed on all object, task, and lesson definitions. Second, the workbook data (the basic lesson and

problem information) is cached, which includes the problem start states and goals. Third, the means-ends facts and randchanges are constructed. Finally, the singular, plural, and special message facts are placed at the end. Facts that are agent-specific are cached alphabetically by agent. The next several sections describe the process used to generate the means-ends facts and the randchanges.

C. GENERATION OF THE RECOMMENDED CLAUSES

The **recommended** clause in means-ends takes a member of the goal that is not true in the current state and "recommends" an operator to achieve the goal. This is the exact purpose that the intended effect provides the objects' primitive operations. Therefore, the recommended clause is a converse map from an intended effect to its primitive operation.

Recommended clauses, however, are among those that must be sorted. This is because METutor's means-ends algorithm gives higher priority to the recommended clauses at the top of the Prolog database. This is how, given two or more goal members not true in the current state, METutor determines which operation is the best given the current situation. MEBuildCMP uses a partial ordering scheme to determine the order based on the following rules:

- The clause recommending operator X precedes operator Y if X precedes Y in a task.
- If operators X or Y are used more than once in a task, then the ordering is based on the first occurrence of the operator in the task.
- If operators X and Y are applied in reverse order among multiple tasks, then the ordering is arbitrary and will be based on the other operators in the lesson.
- Operators not used in any task go to the bottom of the database.

D. GENERATION OF THE PRECONDITION CLAUSES

The **precondition** clauses are more complicated than the **recommended** clauses because the primitive operation can have preconditions specified from four difference sources. These sources are referred to by type, producing Type I preconditions through

Type IV. If an operation is not used in any task in the lesson, then only Type I and Type II preconditions apply. If an operation is used in the lesson, then all four preconditions apply.

- A *Type I* precondition is an explicit precondition described in the primitive operation.
- A *Type II* precondition is an implicit precondition of the primitive operation. It is the "opposite" state of the intended effect. This is a precondition because otherwise the operation would have no effect.
- A *Type III* precondition is an explicit precondition provided in the task definition. Rarely will any operation have Type III preconditions.
- A *Type IV* precondition is an implicit precondition based on the ordering of actions a task. The intended effect of the previous action becomes a precondition of the operation. Often the Type II and Type IV preconditions will be the same.

In addition, **preconditions** may be subject to context. This only applies if one operation is used more than once within the same task. The context helps determine which application of the operation corresponds to which **precondition** clause. The context is determined by taking the guaranteed state in which each occurrence of the operation exists and comparing them. Those items in the state that are guaranteed to differ become the context. A null context argument means that the clause applies to all applications of the operation.

Precondition clauses are also sorted items. The sorting is based on the desire to access the most restrictive **precondition** clause first. Restrictiveness in this sense is defined as the number of elements in the context argument. Longer contexts are placed first. Null contexts are placed last.

E. GENERATION OF THE POSTCONDITION CLAUSES

The **addpostcondition** and **deletepostcondition** information come from two sources -- the objects' primitive operations (*Type I*) and the task operations (*Type II*). The postconditions from the primitive operations consist of the intended effect plus the side effects. The postconditions from the task are the definite side effects only. Probabilistic side effects are treated differently because their effect is not guaranteed. Once the

postconditions have been collected, they make up the addpostcondition information and the opposite of each addpostcondition member makes up the deletepostcondition.

Because operations may be used more than once in a task and therefore may carry different side effects, these clauses also have context arguments. However, the context argument is only non-emtpy for those operations with task-defined side effects. In addition, if the task-defined side effects are identical for all uses of the primitive operation, then the postconditions are merged together with a null context.

Postcondition clauses are sorted in the same manner as precondition clauses. The longer context arguments go to the top of the database and are accessed first.

F. GENERATION OF THE RANDCHANGE CLAUSES

The **randchange** or random-event clauses come in many different forms. For this reason, **randchanges** are also given Type designations.

- A *Type I* random-event is based on uncertainty among members of the initial setting. The teacher specified these in terms of percentages when listing responses to the "condition is probabilistic" sequence of questions.
- A *Type II* random-event is based on the probabilistic side effects given in the task. These random-events are operation-triggered.
- A *Type III* random-event is based on object daemons. *Type IIIa* use probabilities. *Type IIIb* use counts to advance. *Type IIIb* random-events might sound less random then their probabilistic counterpart. However, since these daemon-based events are condition triggered, the advancing event is not guaranteed to occur. Hence,MEBuilder treats them like a random event.

Randchange facts consist of the following information, and are agent-independent. The first item is the triggering action -- for Type I it is **init**, for Type II it is the operation name, for Type III, it is **any_op** to represent "any operation". The second is the context, which is calculated the same way as with precondition clauses. Context arguments are only non-null in a Type II **randchange**. The next two arguments are the postconditions -- delete and add. The fifth argument is the probability of occurrence or the countdown to

occurrence, discussed further below. Finally, the sixth argument is the message which is printed to the user when the random-event occurs. The message for a Type I is blank.

Type IIIb **randchanges**, based on a countdown to next occurrence, introduce information to the state which is hidden from the user. METutor will maintain a special state member which contains the **randchange**'s postconditions, message, and countdown value. The student is not informed that the countdown is active. After each student turn, METutor will decrement the counter. Once the countdown reaches zero, the postconditions are activated and message passed to the student. Countdowns are the first random-event handled after the student's action.

VI. EXPERIMENTAL RESULTS

During the Summer Quarter of Academic Year 1994, an experiment was conducted to demonstrate that MEBuilder's method of lesson authoring was more robust and less time consuming than authoring a lesson in a traditional Computer-Aided Instruction (CAI) form. Appendix F contains all the information disseminated and gathered during the experiment.

A. PARTICIPATION IN THE EXPERIMENT

The experiment included six students, hereafter referred to as the "participants" taking the Advanced Artificial Intelligence class at the Naval Postgraduate School. All six have taken an introductory artificial intelligence class during the spring quarter. The students had never used METutor before. As part of the advanced course content, the participants received some basic instruction about CAI methods and introductions to intelligent tutoring systems.

The participants are American military officers. None had ever authored a lesson for an intelligent-tutoring system. All have experience as military trainers, but most have little or no teaching background. Therefore, the experiment will not target how well MEBuilder works in an actual educational setting. Rather, it will focus on MEBuilder's ability to outperform CAI in terms of simple lesson construction -- does the task of building a lesson take less time, is it more complete, and does it produce fewer errors?

B. SCOPE AND CONDUCT OF THE EXPERIMENT

1. The Participants and Their Requirements

Tab 1 of Appendix F contains the detailed instructions given to the students. The participants were divided into two groups, but each participant was to work individually. The first group of four participants was tasked to write a lesson for a scuba diver preparing to dive for lobster (see Tab 2, Appendix F). The second group of two participants was to

write a lesson for replacing a gasket in a car engine's water pump (Tab 3, Appendix F). The reason for the imbalance is because two participants had to withdraw from the experiment and there was insufficient time to realign the groups. The two tasks were selected and modified such that:

- Both tasks required 14-15 steps, so the amount of work is similar among the two groups.
- Both tasks have 36 possible solutions.
- Both tasks were originally written for METutor versions 21-27 and are ideal tasks for a CAI-based tutoring system.

The participants were provided with access to MEBuilder and METutor, along with CAIBuilder and CAITutor -- a lesson authoring system built with CAI methods and a CAI-based tutoring system. CAIBuilder is written on top of CAITutor in the same manner as the MEBuilder system in order to duplicate the authoring-to-shell environment. Finally, the participants were given access to automated measurement tools which helped collect some of the required data. Tab 4 of Appendix F describes how the participants were to use the automated tools.

In order to provide as fair a comparison as possible between the two methods, several restrictions had to be placed on use of MEBuilder -- specifically those features which the CAI method clearly has no equivalent. For example, the lesson was to be written using one and only one task. This is because CAIBuilder does not have a mechanism of combining tasks into a single task. Second, the students were only to build one problem in the lesson workbook frame as part of the experiment. CAI has no equivalent to MEBuilder's workbook frame. The features of MEBuilder not tested in this experiment will be tested in future. Third, the order of use between the CAI method and MEBuilder was mixed -- four students used the CAI method first, two used MEBuilder first. Again, the imbalance was due to the withdrawals.

Finally, there was a six-hour ceiling on the experiment. Any participant reaching six hours was to stop and turn in the partial results. This was due to time constraints on the availability of the participants. In order to help meet the time constraint and still adequately

test the task-manipulation processes of both methods, the students were provided with partial solutions. These partial solutions, given in Tabs 5 and 6 of Appendix F, contained a task structure of one complete solution with no options.

2. The Data to be Collected

The information the students were required to gather included time spent using each method, number of operations using each method, and some statistical measurements on the resulting data structures. They also had to answer some questions regarding how their time was spent using the CAI and MEBuilder methods. Even though some of these measurements are numeric, they were not intended to be interpreted as significant raw data. Instead, as described below, these measurements were to be interpreted subjectively as a means of identifying trends. With the exception of time, all the precise measurements were done through automated means, as described in the following sections.

a. Time Measurements

There were two measurements requested -- the amount of time spent on the CAIBuilder and MEBuilder programs, and a subjective breakdown of how the time was spent. The time is to be given in hours, and is not intended to be a precise measurement. Rather, it is a subjective measurement to see if one method was significantly quicker than the other. The two tasks were written in such a way that if done properly the students should spend roughly an equivalent amount of time on each solution. The students were specifically instructed not to include down time due to program bugs.

For the second part, the students had to rank the amount of time spent in the following four processes: familiarizing with the program, designing the data, entering the data, and testing and using the data. These were given for CAIBuilder and MEBuilder as a whole, and then asked separately for the class, task, and lesson layers within MEBuilder. Familiarization encompassed reading the user's manuals and running the samples given inside. Designing the data meant organizing the data on paper prior to running the authoring system. Entering the data encompassed time spent using the adding commands

to get the data into the system. Testing the data encompassed checking, debugging, and modifying the data once it was in the system.

The expected trends were that CAIBuilder users would identify entering and testing as two most time-consuming processes. MEBuilder users, on the other hand, would identify familiarization and design at all layers with the possible exception of entering at the object layer. These would indicate that MEBuilder has a steeper learning curve. However, if the time spent on MEBuilder was less, then that means the productivity using MEBuilder is significantly greater.

b. Operations Measurements

CAIBuilder and MEBuilder were embedded with a counting mechanism in order to determine how many commands were used in each system and the percentage of commands aborted. An aborted command is a command that failed or a command whose subsequent queries were explicitly aborted by a participant. In addition, MEBuilder's commands were broken down by layer. These help support the time measurements above by identifying interface problems as a possible factor. If a command is confusing or a participant doesn't understand what the queries mean, he will tend to abort the command. On the other hand, if a particular layer was identified as time consuming in the entering and testing processes and it had few aborted commands, that would signal that the layer's interface is inefficient.

c. Examining the Data Structures

Examining the resulting data structures would provide indicators about the probable correctness of the resulting tasks and how robust the task is. Since the operator names were not strictly standardized, an automated examination of the content of the structures is not feasible. However, the number of acyclic paths to the solution can be measured efficiently. If the participant's tasks are in accordance with the lesson specifications given, his lessons will have 36 solutions. Because CAI methodology puts the

burden on the participant to construct the paths, it was anticipated that the CAI solution counts would vary more than the MEBuilder solution counts.

"Robustness" is in term of the number of nodes and transitions needed in the respective data structures. Since MEBuilder relies on METutor for coaching rules, MEBuilder users do not have to add information about student errors into the task. If the student fails to follow the correct sequence, then METutor will handle the error. This means that the number of nodes and transitions needed by MEBuilder are minimal -- a maximum of one transition and one node per operation.

However, CAIBuilder users must add explicit error states and transitions. For each transition along a solution path, the lesson should provide two or three wrong answers to afford the student a choice. In addition, each wrong answer transition requires a path back to the solution. A non-robust method would have only a single error state and single transition back to the start. A good method would have a single error state per error transition. So a fully robust CAIBuilder solution should have roughly three times the nodes and six times the transitions that the MEBuilder solution might have. Combined with the time and operation measurements, this would help describe how much better MEBuilder could do with a much more complicated task.

3. The Deliverables

In addition to the above data, the participants were required to provide a short write-up of their work. The write-up was to include comments and opinions regarding the experiment and the MEBuilder program. Attached to the write-up is the resulting data files produced by the two authoring systems, along with script files showing the lesson being used in the corresponding tutoring shells. The evaluation of the experiment focused on the write-up, specifically if it included comments about the interface and problems encountered not adequately identified elsewhere.

C. RESULTS OF THE EXPERIMENT

Appendix F contains the raw data generated by the data collection programs on all six participants. The data collection program outputs are in Tab 7 and selected comments from the participants are in Tab 8.

1. Time Measurements

The participants took roughly the same amount of time to do the task on either system, with the CAI system requiring slightly more time. The minimum time spent on CAIBuilder was one hour (by one participant), the maximum was three hours (by three participants), and the average was 2.5 hours. For MEBuilder, the minimum was one hour (by two participants), the maximum was three hours (by one participant), and the average was 2.0 hours. Four of the six required less time to complete the requirements in MEBuilder. Only one required more time with MEBuilder, and the sixth spent an equal amount of time with each system. In both systems, familiarization with the program and entering the data were cited as the most time consuming processes, although MEBuilder showed a greater distribution of time usage than CAIBuilder.

2. Operations Measurements

The participants required from 68 to 164 commands to complete the task in CAIBuilder, with an average of 123. With MEBuilder, however, the range was only 11 to 45 with an average of 26. In terms of completed commands (total commands minus aborted commands), CAIBuilder users required an average of 121 while MEBuilder users averaged 15. The minimum number of commands needed to complete the requirement in CAIBuilder was 30, so CAIBuilder users completed four times the necessary commands. With MEBuilder, the minimum number of commands needed was ten, so the participants performed 1.5 times the necessary commands in MEBuilder.

Participants aborted MEBuilder commands nine times more often than CAIBuilder commands. In CAIBuilder, two participants completed the requirements without having to abort any commands. The highest percentage of aborted commands in

CAIBuilder was 6.5% and the average was just above 2%. In MEBuilder, however, only one participant performed no aborts and two participants had to abort more than 30% of thier commands. The average rate for the participants was 18%.

It is important to note that the data-collection program failed with one participant and the MEBuilder command usage was lost. The participant stated that his command usage was not significantly different from the norm.

3. Resulting Data Structure Measurements

The data structures produced all MEBuilder sessions were identical to the solutions produced by the author. The CAIBuilder solutions, however, differed. All participants stopped at the point where the CAI task was complete, without adding error conditions or states. The four diver-problem participants did achieve the author's solution of 23 nodes and 32 transitions. However, those doing the cooling-system problem did not match the author's resulting data structure nor did they achieve 36 solutions to the task.

4. Comments from the Participants

The positive comments focused on one primary theme. Both systems were deemed simple enough to use once one gets accustomed to them. Neither system was so difficult to use that they felt unable to complete the task. In addition, once the participants became accustomed to MEBuilder they found MEBuilder quicker and more flexible.

The vast majority of the negative comments collected centered on two major themes. The most common concerned the user interface. Both CAIBuilder and MEBuilder use a somewhat crude command-line interface that wasn't friendly and had errors. Most participants agreed that the help facility was weak. Comments specifically directed at MEBuilder was that the sequence of steps in the "create lesson" command were confusing.

The second theme was that once a particular part of the requirements were complete, it was not readily apparant what to do next. For example, once a participant finished with a task, some did not understand that the next step was to construct the lesson.

D. INTERPRETATION OF THE RESULTS

The results show that using MEBuilder's task-manipulation method produced consistent and correct results more quickly than the CAI method. Despite the fact that none of the participants added student-error transitions to the task, they still required more time to complete the task manipulation than with MEBuilder. In addition, CAIBuilder users performed significantly more commands than required due to navigation and editing. This is a clear indication that MEBuilder's method is more efficient.

Further, had the students been required to work with a more complex problem the results would more heavily favor MEBuilder. To illustrate, consider the original lessons from which the tasks used in the experiment were derived. The original cooling-system problem (McDowell, 1993) was 18 steps long but had 720 solutions, not counting the unordered actions. The original scuba-diver problem (Seem, 1992), not counting unordered actions, had 22 steps and 36 solutions.

The CAI data structure for the original cooling-system problem would have required 1080 transitions to model the solutions alone. Since the CAI method works via one transition per command, the rate of command usage would likely have changed little. Therefore, by extrapolating the time and command usage, CAIBuilder users would require 20 hours to build the extended task. One could not expect a teacher to do so and produce an error-free lesson. On the other hand, because MEBuilder allows single commands to perform significant modifications to a structure, users could create the 720 solution in minutes with only a few commands.

With its two unordered actions, the number of solutions to the cooling-system problem increases to 69,192. MEBuilder only requires one command to declare an unordered action, so only two commands are needed to achieve this increased complexity. Clearly, CAIBuilder users would not be reasonable capable of producing an equivalent lesson. The original scuba-diver problem had three unordered actions and 4,320 solutions.

E. CONCLUSIONS

This experiment proves MEBuilder's concept that an authoring system using intelligent rules can produce lesson material much more efficiently than traditional CAI. In addition, the resulting MEBuilder lesson requires far less code space and can be modified significantly faster than with a traditional CAI method.

This experiment also shows that for MEBuilder to be effective, the user interface is extremely important. Such an interface should assist a teacher in entering commands to the terminal through the use of menus and structured dialogs. It should also provide a means of helping the teacher understand the authoring process, and provide a good help facility for the teacher to fall back on.

The experiment also showed that more work is needed in the lesson-definition system. The interface needs to make the overall process more intuitive. An appropriate follow-on experiment would have the participants given a set of tasks and by required to construct different types of lessons. The lessons would contain problems that range in difficulty from beginner-level to expert-level. The levels of difficulty could be achieved by breaking the task into components or adding random hazards to the problem.

VII. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

A. SUMMARY OF CONTRIBUTIONS

MEBuilder's object-oriented design and teacher-assistance tools show great promise in the construction of a general-purpose lesson authoring system. Its library-management features help organize lesson information and hide the low-level file structure from the teacher. Its object-modelling techniques help ensure consistency and reduce errors when building tasks. Its task-modelling techniques help build complex yet robust procedures in a manner of minutes. Its workbook-style lesson framework help a teacher construct a multitude of exercises to serve a wide variety of purposes -- from increasing levels of difficulty to presenting different subtasks.

The experimental results supports MEBuilder's concept and design as having tremendous potential. It showed that MEBuilder could be a more effective and efficient authoring system than one based on traditional CAI methods. Further, MEBuilder produces lessons with a higher degree of assurance and a lower risk of error. Also, because of MEBuilder, METutor has evolved to handle a greater range of problems. These problems include those involving other characters, multiples of like props, and multiple exercises in a lesson.

B. WEAKNESSES OF MEBUILDER

MEBuilder is not without problems, especially given that it is a project in its genesis. There are several areas which require significant adjustments and improvements before this program can be considered releasable.

1. Is Object-Modeling Too Complex For Teachers?

MEBuilder is heavily reliant on object-oriented modeling techniques, which may be beyond the comprehension of many computer-illiterate educators. Currently, the

object-definition module is very unhelpful in its presentation. A teacher might not understand what it means to inherit property sets or operations. Part-kind inheritance is an extremely difficult concept even for most students of artificial intelligence. MEBuilder requires additional features that help visualize the object as it is being defined, and help the teacher understand the implications of modifying object inheritance. In addition, future experiments should be conducted using professional educators outside the realm of computer science. Military trainers would be a good example.

2. Lack of Pre-Defined Object Library

In order for MEBuilder to be effective, it must come with an extensive library of pre-defined objects and tasks. Otherwise, a great number of lessons will consist of objects that are only good for that lesson or whose behaviour has been limited to meet the needs of only one lesson. In addition, the object layer is where the greatest amount of time is spent, and the desire is for the teacher to devote energy mostly at the lesson layer. Object re-use is a major selling point in object-oriented modeling, and only with an extensive and accurate library of objects can this benefit be realized.

3. MEBuilder and METutor Do Not Employ the Same Domain of Features

There are features built into METutor which MEBuilder presently does not access. This includes a wider domain for summary-fact definitions, multiple goals, and other quantifier expressions within the macro languange templates. The converse is also true. The macro language does not perfectly handle class inheritance of objects, so a problem written for a prop of some object might not work exactly right for objects of a derived class. A lot more work is required to ensure that the features of MEBuilder and METutor are brought to a perfect one-to-one correspondence.

4. Emphasis Needed on the Interface

The initial focus on MEBuilder has been on its architecture, algorithms, and data structures. However, as shown in the experiment, MEBuilder requires a strong user-

interface in order to be effective. The user-interface needs to provide easier command access, simple and understandable representations of all MEBuilder entities, and a thorough and context-sensitive help facility. A graphical interface using menus and windows would be ideal.

C. FUTURE RESEARCH DIRECTIONS FOR MEBUILDER AND METUTOR

1. Coaching Capabilities

The scope of both programs only extends to the conduct of exercises and some basic tutoring rules based on means-ends algorithms. In order to help make the resulting lesson more believable to the student, both programs require means of specifying and implementing domain-specific coaching rules. These are rules which can evaluate a full sequence of student actions and help find possible cognitive errors that a means-ends based error will not detect.

2. Including Ancillary Domains in MEBuilder

Some entities cannot be modeled adequately as a property set, but can be modelled using summary information. An example of this is an audit file for an operations system, which contains a list of the operations performed by users in a computer center simulation. The entries of an audit file constitute the file's state, however one certainly cannot efficiently model these entries beforehand in a property set nor use the raw data in means-ends analysis. Summary information based on audit file, on the other hand, can be used in means-ends.

An audit file is an example of an ancillary domain. The term "ancillary" is appropriate because it supplements the state with raw data. The extension of MEBuilder (and therefore METutor) to cover ancillary domains must handle three areas -- the definition of the ancillary domain and the associated summary facts, the definition of operations that impact the ancillary domain, and the relationship of the ancillary domain in the task and lesson structure. Other important but less critical issues include the display of

ancillary data. The qualitative state methodology of METutor lends itself to a simple output interface. Ancillary domains would require a more complex interface.

3. Including Quantitative Domains and A* Search Techniques in MEBuilder

MEBuilder is still limited to qualitative problems that only view the state as a list of objects' current properties. Many applications have quantitative values involved, such as reading on a boiler or an operation such as "turn dial to X". The current object layer in MEBuilder can be extended to handle different domains, including quantitative ones, along with associated operations.

In addition, the means-ends algorithm in METutor uses the ordering of facts in the database to break ties among sets of operations that can be performed at a given step. The ordering of facts may be inappropriate or incorrect given some contexts. In addition, this method only helps answer the question of what to do, but not how much to do it. In order to adequately apply quantitative operations in a given problem, the means-ends algorithms in both programs require supplementation with a quantitative search method. A* is suggested here because it is the most general purpose search method available and it is built to address the specific issue of solving a procedural task in the least costly manner (whether in terms of time, resources, etc.). A* as a basis for an intelligent tutoring system has been explored separately (Galvin, 1994, p. 725).

4. Use of MEBuilder in a Major Application

Currently, MEBuilder has only been tested in the context of simple applications. The next set of tests involving MEBuilder must demonstrate its ability to handle complex simulations with many different agents. An example of such an application might be a tutoring system for a system administrator learning computer security. In such an application, legitimate users and intruders would serve as agents in a simulated environment and the student would be charged with maintaining the system. Such an application would take full advantage of MEBuilder's object-oriented modeling and would

be sufficiently complex to test MEBuilder's simulation capabilities. It would also help uncover bugs that simpler applications do not produce.

5. Using MEBuilder with Other Intelligent Tutoring Systems

Currently, METutor is the only intelligent-tutoring system with which MEBuilder will work. In addition, MEBuilder uses many of the same algorithms as METutor during automated task generation. This allows MEBuilder to guarantee that its lesson will work in METutor. However, MEBuilder's value would be greatly increased if it could generate lessons for other established intelligent-tutoring systems, such as PIXIE (Sleeman, 1987, p. 239). To accomplish this, MEBuilder requires user-selectable Lesson Compilers, one for each supported intelligent-tutoring system.

APPENDIX A. MEBUILDER SOURCE FILES

This appendix only contains the header comments for the primary MEBuilder source files and the primary METutor version 29 source file. This is for practical considerations given that the source is nearly 400 pages long and contains more than one megabyte of text.

Only those source files directly relevant to MEBuilder's primary functions are included. Files not included are the help system source files, several Prolog utility files written by the author, and the supplemental data collection module used for the experiment. Also, the author has deleted segments of the header comments from the below files which describe future upgrade requirements

Tab 1. mebuild.pl -- MEBuilder's Main Module
Tab 2. mebuild_class_definition.pl -- MEBuildCLS module source
Tab 3. mebuild_task_definition.pl -- MEBuildTSK module source
Tab 4. mebuild_lesson_definition.pl -- MEBuildLES module source
Tab 5. mebuild_lesson_compiler.pl -- MEBuildCMP module source
Tab 6. mebuild_library.pl -- MEBuildLIB module source
Tab 7. metutor29_shell.pl -- METutor version 29 source

TAB 1. MEBUILDER MAIN MODULE

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1            (MEBUILDER)    */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Main Interface -- Version 1                                     */
/*                                                                           */
/*    To run MEBuilder, load *this* module and query:                        */
/*                                                                           */
/*         :-  build.                                                        */
/*                                                                           */
/*    NOTE:  To run MEBuilder with the current Prolog database intact, query:  */
/*                                                                           */
/*         :-  build_without_initializing.                                   */
/*                                                                           */
/*    The latter is designed for use as a recovery tool should MEBuilder perform */
/* a less-than-graceful exit during execution.                               */
/*                                                                           */
/*    The main interface module provides the command loop and access to the  */
/* subordinate interface modules.  It also manages the autosave file -- which is */
/* a local file dump of all dynamically created modules -- and the help facility */
/* -- which is a simple sub-command loop providing somewhat context sensitive  */
/* help to the user.  Access functions to the subordinate data structures are  */
/*provided through the following facts:                                      */
/*                                                                           */
/*    legal_command(<command>,<predicate>).                                  */
/*                                                                           */
/*    <command> is the input retrieved from text_io's get_prompted_input.  The  */
/* predicate is a zero-argument predicate which performs the analagous command.  */
/* legal_command is a multifile fact, but is *not* dynamic.                  */
/*****************************************************************************/
```

TAB 2. MEBUILDER CLASS MODULE (MEBuildCLS)

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1            (MEBUILDER)    */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Class Definition Module -- Version 1.01                         */
/*                                                                           */
/* Version History                                                           */
/*  1.0   -- First release. concentrates on component, property set, and     */
/*           operation definitions.  The relations and daemons are defined and */
/*           included but their effects are far from guaranteed to work in a */
/*           compiled lesson.                                                */
/*  1.01  -- Comments are updated and stubs placed for commands or predicates */
/*           needed for use in future.  The property_display_data and the    */
/*           operation_display_data sections, which were not used in the     */
/*           compiler, are stubbed out.  (These should be fixed before I go). */
/*                                                                           */
/* Important Compilation Note:                                               */
/*   During the Quintus Prolog linking process, the following two predicates are */
/*   announced as unknown.  This is caused by compiling and linking the program */
/*   in the Prolog, not the Prowindows environment:                          */
/*         user:prowindows/0                                                 */
/*         user:draw_property_picture/0                                      */
/*         user:tbd/N      (for any N -- used to identify code stubs)        */
/*   If and when the mebuild_class_definition_graphics file is permanently in- */
/*   stalled into MEBuilder, these messages will go away.                    */
/*                                                                           */
/*                                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's Class definition data structure                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                           */
/*    A class is defined as the composite of the following facts with the same */
/* <class> argument as the first argument.                                   */
/*                                                                           */
/*    class_def(<class>, <list of parent classes>).                          */
/*       NOTE on <parent classes>.  The standard parent classes of prop and  */
/*       character are used to indicate the specific type of object it is and */
/*       what capabilities it has.  Character objects have the additional cap- */
/*       abilities of being the primary actor of a task, and can be instantiated */
/*       as the student's role in a lesson.  Currently only single inheritance */
/*       is supported by MEBuilder, however the inheritance facilities provide */
/*       for expansion into multiple inheritance.                            */
/*                                                                           */
/*    component(<class>, <component class name>, <component name>, <tense>).  */
/*       NOTE on components:  Tense is given as singular, plural, or default and */
/*       is used to override the "ends in s" rule when determining plural nouns */
/*       in the standard English-language output.                            */
/*                                                                           */
/*    property_set(<class>, <property set name>, <domain>, <hideable>).      */
/*       where <hideable> ::= hideable | not_hideable                        */
/*       <hideable> corresponds to the creation of hideable facts in the METutor */
/*       lesson.  A property set that is hideable automatically generates holes */
/*       for the specification of operators that handle hideable facts.      */
/*       NOTE on property_sets:  Past implementations of MEBuilder used the same */
/*       structure for members of a property set as for the METutor facts used */
/*       a means-ends search space.  This version used only the prefix, which */
/*       is intended to eliminate the wasteful operand replacement.          */
/*                                                                           */
```

```
/*    property_display_data(<class>,<property>,<graphics data>).                */
/*       For version 1.01, all interfacing with property_display_data has been  */
/*       deleted and stubbed out.  The mebuild_class_definition_graphics file   */
/*       contains the capability for specifying a graphic for the property.  The */
/*       <graphics data> would contain only those things that are generically   */
/*       true, such as the graphic bitmap file and the dimensions.  All other   */
/*       details, such as the color, location, click range, etc. are part of the */
/*       individual lesson definition (refer to lesson definition file.         */
/*                                                                              */
/*    relation(<class>, <object>, <property>, <definition set of properties>).  */
/*       A relation is a meta-property which is true if the <definition set of  */
/*       properties is met.  <object> is either <class> or a valid component,   */
/*       and comprises the state argument to <property>.  <property> is a symbol */
/*       and <definition set> is of the form atomic -- corresponding to a       */
/*       property of the class, or of the form <property>(<component>) for those */
/*       properties held for the component.                                     */
/*                                                                              */
/*    daemon(<class>,<daemon name>,<triggering conditions>,<advancement cond>,   */
/*           <advance type>, <message template>).                               */
/*       where <triggering conditions> are conditions that cause the daemon to  */
/*       "wake up" or "stay awake".  The daemon is checked at each turn when the */
/*       <triggering conditions> are true.  When a daemon is activated, it is   */
/*       doctrinally set to the first member of the iterating property set,     */
/*       although in means-ends space it can in theory start at any place.      */
/*       <advancement cond> are special conditions that indicate that the daemon */
/*       will wake up -- either prob(<probability>) or count(<integer>) where   */
/*       integer corresponds to the number of turns.  The <advance type> cor-   */
/*       esponds to one of two things -- either advance(<property set of class>) */
/*       which indicates that the daemon will always advance to the next        */
/*       property in the set, loop(<property set of class>) which the same as   */
/*       advance except that the object's value can revert to the first value,  */
/*       or update(<operation>) where <operation> is a defined operation of the */
/*       class which takes *no* indirect objects.  (This is probably an unneeded */
/*       restriction which could be cured in future.                            */
/*       The message template is used when the daemon becomes active.  Else,    */
/*       the messages used to describe the daemon's advancement fall in line    */
/*       with the property set members and the <operation>'s apply text.        */
/*                                                                              */
/*    operation(<class>, <object list>, <verb>, <direct obj.>, <trail phrase>,  */
/*              <precondition list>, <intended effect>, <side effect list>).    */
/*       NOTE:  The 4-tuple (<object list> <verb> <direct obj.> <trail phrase>) */
/*              is considered as a whole to be the identifying name of an op.   */
/*       operation models specifically one type of method -- the atomic method. */
/*       operation facts describe an atomic operation -- one that takes one     */
/*       agent-turn to perform and cannot be broken down further.  operations   */
/*       are not character independent -- they can be overloaded by specifying  */
/*       the highest level character class as an object.  If no character class */
/*       is specified, then there is an implicit assumption that all character  */
/*       classes can perform the operation.                                     */
/*       <class> is considered the direct object's class.  <object list> is a   */
/*       complete list of objects that are required to be present in order to   */
/*       perform the operation.  Note that class can be repeated -- meaning that */
/*       an operation must take two or more distinct objects of the same class. */
/*       There will exist a means to specify forall in operations however such  */
/*       capability is not yet present.                                         */
/*       The <verb> is a verb phrase.  <direct obj.> is either <class> or a     */
/*       valid component of <class>.  <trail phrase> can be any combination of  */
/*       words, however any subsequence of words which match a member of        */
/*       <object list> will be functored with obj() in order to do inheritance  */
/*       on the name of the operation.  <precondition list> matches one-for-list */
/*       with respect to [<class>|<object list>].  <intended effect> is a single */
```

```
/*       property which will become true upon application of this operation and  */
/*       must be a defined property of <direct obj.>.                             */
/*                                                                                */
/*    operation_display_data(<class>, <operation>, <display data>).               */
/*       For version 1.01, all operation_display_data items have been deleted or  */
/*       stubbed out.  The only useful thing for <display data> would be the      */
/*       generic or default text string used to generate an apply_text in a       */
/*       lesson.  <display data> could be expanded to include animation data      */
/*       for showing an operation occurring.  This is not used as a 9th arg to    */
/*       the operation fact because the hierarchy of display items does not       */
/*       necessarily follow that of the preconditions and postconditions.         */
/*                                                                                */
/*                                                                                */
/*    Classes are modeled using Quintus Prolog dynamically-created modules, so    */
/* the facts for a class C is C:class_def(C,PCL) - C:component(C,CC,CN) - etc.    */
/* This allows this module to take advantage of the module feature in text_io's   */
/* prolog_outfile predicate.                                                      */
/*    Classes are not directly accessed by outside modules.  cache_class_facts    */
/* is used to take all the class information and cache it into data retrievable   */
/* by the lesson database.                                                        */
/*                                                                                */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Commands provided by this module (including those not yet implemented)         */
/* Documentation on these commands can be found in the user's manual.             */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                                */
/* OBJECT DEFINITION COMMANDS:                                        LOOPS:      */
/*    create object [named <object>]                                  Main        */
/*    remove object [named <object>]                                  Main        */
/*    restore object [named <object>]                                 Main        */
/*    view object [named <object>]                                    All         */
/*    modify parent object [of <object>]                              Main        */
/*    check object [named <object>]                                   All         */
/*                                                                                */
/* COMPONENT MANAGEMENT COMMANDS:                                                 */
/*    create component [named <component>] [for <object>]             Main        */
/*    remove component [named <component>] [from <object>]            Main        */
/*    view component [named <component>] [of <object>]                All         */
/*    modify component [named <component>] [of <object>]              Main        */
/*                                                                                */
/* PROPERTY SET MANAGEMENT COMMANDS:                                              */
/*    create property set [named <property set>] [for <object>]       Main        */
/*    remove property set [named <property set>] [from <object>]      Main        */
/*    view property set [named <property set>] [of <object>]          All         */
/*    modify property set [named <property set>] [of <object>]        Main        */
/*    set property display data [for <property>] [of <object>]        Main        */
/*                                                                                */
/* SUMMARY FACT MANAGEMENT COMMANDS:                                              */
/*    create summary fact [for <object>]                              Main        */
/*    remove summary fact [from <object>]                             Main        */
/*    view summary fact [of <object>]                                 All         */
/*    modify summary fact [of <object>]                               Main        */
/*                                                                                */
/* OPERATION MANAGEMENT COMMANDS:                                                 */
/*    create operation [for <object>]                                 Main        */
/*    remove operation [from <object>]                                Main        */
/*    view operation [of <object>]                                    All         */
/*    modify operation [of <object>]                                  Main        */
/*    set operation display data [of <object>]                        Main        */
/*                                                                                */
/* BACKGROUND CHANGE MANAGEMENT COMMANDS:                                         */
```

```
/*    create background change [named <daemon>] [for <object>]        Main       */
/*    remove background change [named <daemon>] [from <object>]       Main       */
/*    view background change [named <daemon>] [of <object>]           All        */
/*    modify background change [named <daemon>] [of <object>]         Main       */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported Predicates.  All predicates intended for external use are prefixed   */
/*  with "mebuildCLS_"                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* APPLICATION PREDICATES (All exported to mebuild main):                        */
/*    mebuildCLS_setup.                                                          */
/*        -- Initializes all the domains, commands, and templates for the main   */
/*           mebuild application command loop.                                   */
/*    mebuildCLS_include_view_commands(+Loop)                                    */
/*        -- Provides Loop with the class definition view commands so use in     */
/*           the task definition and lesson definition loops.                    */
/*    mebuildCLS_class_database_initialize.                                      */
/*        Initializes the class_definition database.                             */
/*    mebuildCLS_class_database_shutdown.                                        */
/*        Clears all class_definition facts from the Prolog database and erases  */
/*        all dynamically-created modules.  Used by MEBuild's quit command.      */
/*    mebuildCLS_append_autosave.                                                */
/*        Appends all class definition data in memory to the autosave file.      */
/*                                                                               */
/* CLASS DEFINITION MANAGEMENT:                                                  */
/*    (COMMAND INVOKED)                                                          */
/*    mebuildCLS_define_class(+Class).                                           */
/*        Provides user access for defining a new class.                         */
/*    mebuildCLS_remove_class(+Class).                                           */
/*        Marks a class as "deleted" (refer to library module).                  */
/*    mebuildCLS_restore_class(+Class).                                          */
/*        Restores a removed class in memory (i.e. undeletes it).                */
/*    mebuildCLS_view_class(+Class).                                             */
/*        Pretty-prints the class definition to the screen.  Only provides a by- */
/*        name listing of the class members.  For more detailed information a-   */
/*        bout a particular data item, refer to the corresponding view command   */
/*        below.                                                                 */
/*    mebuildCLS_modify_class(+Class).                                           */
/*        Allows the user to change the parent class link.                       */
/*    (COMMAND INVOKED AND EXPORTED TO OTHER MODULES)                            */
/*    mebuildCLS_test_class_integrity(+Class).                                   */
/*        Allows the user to test the integrity of the class to ensure the class */
/*        definition is consistent.  The test includes checking whether or not:  */
/*           -- All referenced components, properties, and operations in the     */
/*              are still well-defined.                                          */
/*           -- That changes to the well-defined members of the class does not   */
/*              cause other members of the class to become contradictory.        */
/*        The semantics of the particular tests are given in the predicates in   */
/*        each segment named <object>_violates_integrity.                        */
/*    (EXPORTED TO OTHER MODULES)                                                */
/*    mebuildCLS_cache_task_facts(+TargetModule,+List of Instance/Class pairs)   */
/*        Produces cached versions of the class definition for use in the task   */
/*        modules.  Only the property sets and the operations are cached for     */
/*        use in task definitions.                                               */
/*    mebuildCLS_cache_lesson_facts(+TargetModule,+List of Instance/Class pairs) */
/*        Produces cached versions of the class definition during compilation.   */
/*    mebuildCLS_export_classes(+List of Class/File pairs)                       */
/*        Sends out the class definitions in memory to the files listed.         */
/*    mebuildCLS_derived_class_of(+Class,?Descendant)                            */
/*        Tests for derived class relationships or provides a derived class.     */
```

```
/*    mebuildCLS_is_a_prop_class(+Class)                                    */
/*    mebuildCLS_is_a_character_class(+Class)                               */
/*        Specific instances of mebuildCLS_derived_class_of for the generic */
/*        "prop" and "character" objects.                                   */
/*                                                                          */
/* COMPONENT DEFINITION MANAGEMENT:                                         */
/*    (COMMAND INVOKED)                                                     */
/*    mebuildCLS_define_component(+Component,+Class).                       */
/*        Provides user access to the component definition facility.  The   */
/*        Component must be unique to Class.  The user will be prompted separ- */
/*        ately for a Component Class with which the Component will be an in- */
/*        stance of, and it will establish a part_of inheritance relationship. */
/*        Circular part_of inheritance is not allowed and ComponentName must be */
/*        unique.                                                           */
/*    mebuildCLS_remove_component(+Component,+Class).                       */
/*        Undefines the component from the class.  The component must be a  */
/*        defined component of the class, not an inherited one.             */
/*    mebuildCLS_view_component(+Component,+Class).                         */
/*        Displays the entire component definition in pretty-printed form --  */
/*        including the source of the component definition (whether defined in */
/*        the object, inherited from a parent object, or inherited from a   */
/*        component object.                                                 */
/*    mebuildCLS_modify_component(+Component,+Class).                       */
/*        Allows the user to modify any of the attributes associated with the */
/*        component definition -- type, name, or tense.  The component must be */
/*        a defined component of the class, not an inherited one.           */
/*    (EXPORTED TO OTHER MODULES)                                           */
/*    mebuildCLS_get_singular_component(+Class,-SingularComponent).         */
/*    mebuildCLS_get_plural_component(+Class,-PluralComponent).             */
/*        Retrieves a component definition of the Class which overrides the */
/*        tense default (singular and plural, respectively).                */
/*                                                                          */
/* PROPERTY SET DEFINITION MANAGEMENT:                                      */
/*    (COMMAND INVOKED)                                                     */
/*    mebuildCLS_define_property_set(+PropertySet,+Class).                  */
/*        Provides user access to the property set definition facility.  The */
/*        Property Set name must be unique to the class.  The user will be  */
/*        queried for a list of property set members (currently only lists of */
/*        symbols are allowable), and if the set is hideable (meaning that the */
/*        set represents a fact in the world which could be unknown.        */
/*        The members of List of Properties must be unique for a class, to  */
/*        include class hierarchical links, unless the Property set is redefin- */
/*        ing an inherited property set.  Object hierarchical links do not  */
/*        require this restriction.                                         */
/*    mebuildCLS_remove_property_set(+PropertySet,+Class).                  */
/*        Undefines the property set and undefines all of the member property */
/*        data.  It does not automatically undefine all of the operators and */
/*        relations, etc. (that is extremely complex and it is basically left */
/*        for the teacher to manage using the integrity checker).  The property */
/*        set must be a defined member of the object, not an inherited one. */
/*    mebuildCLS_view_property_set(+PropertySet,+Class).                    */
/*        Prints out a detailed definition of the property set, including where */
/*        it is defined from (either in the object itself, inherited from a */
/*        parent or ancestor, or inherited from a component).               */
/*    mebuildCLS_modify_property_set(+PropertySet,+Class).                  */
/*        Allows the user to modify the property set, either its name, members, */
/*        or its hideability declaration.  The Property set given must be a */
/*        defined property set of the object, not an inherited one.         */
/*    (EXPORTED TO OTHER MODULES)                                           */
/*    mebuildCLS_get_hideable_property_set(+Class,-PropertySet,-Domain).    */
/*        Locates and returns a hideable property set for the compiler.     */
```

```
/*    mebuildCLS_adjacent_properties(+Class,+PropertySet,loop|no_loop,          */
/*                                   ?Property1,?Property2).                    */
/*        For property sets intended to be interpreted as a sequence of states,  */
/*        this function serves as a successor and predecessor function which is  */
/*        capable of returning all pairs if the last two are unbound.  This      */
/*        function is strictly non-transitive.  The loop marker in the third     */
/*        argument indicates that the first member of the set is considered the  */
/*        successor of the last.                                                 */
/*    print_current_property(+Object,+Fact,-Output).                             */
/*        Takes the object and fact and assembles a list of words Output which   */
/*        with print_noun can be outputted.  print_current_property is intended  */
/*        for use with print_list as a template.                                 */
/*    assemble_properties(+Object,+PropertyDomain,-AssembledProperties).         */
/*    xassemble_properties(+Object,+PropertyDomain,-AssembledProperties).        */
/*    yassemble_properties(+Object,+PropertyDomain,-AssembledProperties).        */
/*    assemble_property(+Object,+Property,-AssembledProperty).                   */
/*    xassemble_property(+Object,+Property,-AssembledProperty).                  */
/*    yassemble_property(+Object,+Property,-AssembledProperty).                  */
/*        These are used to put together and disassemble properties to and from  */
/*        their object and basic property components.  Their differences are as  */
/*        follows:                                                               */
/*           plain  = only for objects with their defined properties.            */
/*           x      = for assembling objects with their defined or component     */
/*                    properties                                                 */
/*           y      = for assembling templates of properties (especially for the */
/*                    lesson compiler)                                           */
/*    opposite_of(+Object,+Property,+Opposite).                                  */
/*    negation_of(+Object,+Property,+Negation).                                  */
/*    list_opposite_of(+Object,+Properties,+Opposites).                          */
/*    list_negation_of(+Object,+Properties,+Negations).                          */
/*        Opposites and negations are two concepts that are similar.  For a set  */
/*        of one or two members, they are the same.  For three+, however, the    */
/*        negation of a property X is simply not(X).  The opposite of property X */
/*        in domain oneof([X,Y1,...,Yn]) is Y1, then Y2, ..., Yn.                */
/*        The list versions return bagofs all possible answers and returns it in */
/*        a single-level list.                                                   */
/*    build_ordinal_dual_arglist(+Objects,-PairsofInstance/Objects).             */
/*        Ordinal dual arglists are used to specify an abstract set of instances */
/*        for use in class definitions and tasks.  It takes a list of objects    */
/*        and attaches ordinal names to repeated instances of the objects.       */
/*        Ex. [a,b,c,a,b,a] becomes [a,b,c,[[second,a],a],[second,b],[third,a]]  */
/*                                                                               */
/* PROPERTY DISPLAY DATA MANAGEMENT:                                             */
/*    (COMMAND INVOKED)                                                          */
/*    mebuildCLS_set_property_display_data(+Property,+Class).                    */
/*        Invokes the property display data management subcommand loop.  This is */
/*        mostly stubbed out, but does contain hooks to access the graphics      */
/*        module.                                                                */
/*                                                                               */
/* RELATION DEFINITION MANAGEMENT:                                               */
/*    (COMMAND INVOKED)                                                          */
/*    mebuildCLS_define_relation(+Class).                                        */
/*        Provides user access to the relation definition facility.  The name of */
/*        the relation must be unique to the class.  The user will be queried    */
/*        for the name of the relation (or "summary fact") and then be given a   */
/*        list of choices for the definition (which is a listing of all the      */
/*        defined and inherited properties of the class.                         */
/*    mebuildCLS_remove_relation(+Class).                                        */
/*        Undefines a relation.  The user is given a menu of relations to choose */
/*        from, which will only include defined members of the object -- not     */
/*        inherited ones.                                                        */
```

```
/*    mebuildCLS_view_relation(+Class).                                    */
/*        Prints out a full definition of a relation ("summary fact") to the */
/*        terminal.  The user does not specify the summary fact on the command */
/*        line since parsing a summary fact name is difficult.  Rather, the user */
/*        will be given a menu of all summary facts to select.            */
/*    mebuildCLS_modify_relation(+Class).                                  */
/*        Allows the user to change the name or definition of the summary fact. */
/*        The summary to be edited will be selected from a menu in response to */
/*        the "modify summary fact" command -- it will not be a menu choice. */
/*        This operation is restricted to defined members of the object, not */
/*        inherited ones.                                                 */
/*    (EXPORTED TO OTHER MODULES)                                          */
/*    mebuildCLS_get_partial_relation_definition([+Name,+Class],-Def)      */
/*        Extension of mebuildCLS_define_relation for task-level summary facts. */
/*    mebuildCLS_get_summary_fact(+Class,-SummaryFact,-Definition)         */
/*        Extracts an object-based summary definition.                    */
/*                                                                        */
/* OPERATION DEFINITION MANAGEMENT:                                       */
/*    (COMMAND INVOKED)                                                   */
/*    mebuildCLS_define_operation(+Class).                                */
/*        Provides user access to the operation definition facility.  The name */
/*        of the operation must be unique to the class.  The user will be */
/*        queried for the following information:                          */
/*          Name -- The operation name must be of the form <prefix> <direct */
/*             object> <trail phrase>.  The direct object must be the class name */
/*             or a valid component name of the object.                   */
/*          Associated Objects -- Effectively an argument list for the    */
/*             operation (think of them as the direct and indirect objects). */
/*          Preconditions -- Provide the preconditions for all objects in the */
/*             operation.                                                 */
/*          Intended Effect -- The single intended purpose for performing the */
/*             operation.  It is a property of the direct object.         */
/*          Side Effect -- The other changes to the objects in the operation. */
/*             This list must be a strictly determinate list (no possibilities */
/*             or probabilities allowed here.                             */
/*    mebuildCLS_remove_operation(+Class).                                */
/*        Undefines an operation.  The user is given a menu of operations to */
/*        undefine.  You may only remove defined operations of the object, not */
/*        inherited ones from parents or components.                      */
/*    mebuildCLS_view_operation(+Class).                                  */
/*        Prints out the full details of the operation definition.  It includes */
/*        all the information specified about in mebuildCLS_define_operation, */
/*        plus information about whether this operation is defined in the Class, */
/*        was inherited from a parent or ancestor, or was inherited from a */
/*        component of the object.  The operation is chosen from a menu provided */
/*        through this command -- not by a command-line entry.           */
/*    mebuildCLS_modify_operation(+Class).                                */
/*        Modifies the operation definition.  The operation is chosen from a */
/*        menu invoked by the command -- not given on the command line, and only */
/*        allows the modification of defined operations, not inherited ones. */
/*        The modify operation command allows for the change in ONLY the  */
/*        intended effects, preconditions, and side effects.  You CANNOT rename, */
/*        nor respecify the indirect objects.  These are temporary changes due */
/*        the extreme complexity involved, plus the fact that it would be easier */
/*        for these two if you use the "remove operation" command and then start */
/*        over with the "create operation" command.                       */
/*                                                                        */
/* OPERATION DISPLAY DATA MANAGEMENT:                                     */
/*    mebuildCLS_set_operation_display_data(+Property,+Class).            */
/*        Invokes the operation display data management subcommand loop.  This */
/*        is mostly stubbed out, but does contain hooks to access the graphics */
```

```
/*        module.                                                              */
/*                                                                             */
/* DAEMON DEFINITION MANAGEMENT:                                               */
/*    (COMMAND INVOKED)                                                        */
/*    mebuildCLS_define_daemon(+Daemon,+Class).                                */
/*        Provides user access to the daemon definition facility.  The user will */
/*        be queried for the following information:                            */
/*          Triggering Condition -- What will cause the daemon to become active. */
/*          Activation Type -- Whether or not the daemon will advance on or loop */
/*             on a property set, or its activation is based on applying an    */
/*             operation.                                                      */
/*          Advancement Criterion -- Whether the daemon will advance based on  */
/*             numbers of turns or a probability.                              */
/*          Activation Message                                                 */
/*    mebuildCLS_remove_daemon(+Daemon,+Class).                                */
/*        Undefines a daemon.  Must be a defined daemon, not an inherited one. */
/*    mebuildCLS_view_daemon(+Daemon,+Class).                                  */
/*        Prints out all of the relevant detailed information about the daemon, */
/*        including whether it is defined within the Class, inherited from a   */
/*        parent class, or inherited from a component.                         */
/*    mebuildCLS_modify_daemon(+Daemon,+Class).                                */
/*        Allows the user to modify some of the attributes of the daemon.  The */
/*        daemon must be an defined daemon, not an inherited daemon from either */
/*        the parent or component hierarchy.                                   */
/*    (EXPORTED TO OTHER MODULES)                                              */
/*    mebuildCLS_get_background_fact(+Class,-Trigger,-Prob,-Type,-Msg)         */
/*        Retrieves information about a background fact of the Class.          */
/*                                                                             */
/*******************************************************************************/
```

TAB 3. MEBUILDER TASK MODULE (MEBuildTSK)

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1            (MEBUILDER)    */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Task Definition Module -- Version 1.01                          */
/*                                                                           */
/*   This module manages the task data structures and assists the user in buil- */
/* ding consistent task definitions.  Currently, the task structure uses a   */
/* simple procedural net, covered in the latter stages of this file.         */
/*                                                                           */
/*   Version History                                                         */
/*    1.0   -- Original version released to the students for the experiment. */
/*    1.01  -- Comments are updated and stubs placed for commands or predicates */
/*             needed in future.                                            */
/*                                                                           */
/*   Important Notes about the Current Version:                              */
/*    +The task definition module contains several submodules, each of which */
/*     could (should) be broken out into a separate file.  At present, however, */
/*     the communications among the submodules is too tightly woven to make a */
/*     clean break.  The submodules would be:                                */
/*       -- The MEBuilder interface portion                                  */
/*       -- The procedure graph (or procedural net) manager                  */
/*       -- The guaranteed state (or situation) manager                      */
/*    +The task definition module also makes too liberal a use of utility pred- */
/*     icates from the class definition module. (There are some ten predicates */
/*     that are "exported" in mebuildCLS that do not begin with mebuildCLS_). */
/*    +Although a considerable amount of work has been devoted to modularizing */
/*     MEBuilder, the fact remains that the intramodular structure of this   */
/*     module leaves a bit to be desired.  The procedure graph, guaranteed   */
/*     state, and other submodules need to better conform to the specified or */
/*     intended interfaces.                                                  */
/*                                                                           */
/*                                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's task fundamentals                                             */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                           */
/*   A task establishes a temporal relationship among operations when the oper- */
/* ations are applied towards a specific goal.  This allows the teacher to i- */
/* dentify relationships between and among objects that the basic operations */
/* themselves don't cover.                                                   */
/*                                                                           */
/* Presentation of a task to the user:                                       */
/*   Currently, a task is given as a sequence of operations or sets of sub-  */
/* procedures.  Each operation is given a step number which is printed in front */
/* of the operation.  This step number is used to identify steps when being  */
/* manipulted about the task (rather than having the teacher type the entire */
/* operation name -- which could be ambiguous anyhow since an operation could be */
/* used more than once.)                                                     */
/*       [1]  turn the key                                                   */
/*       [2]  open the door                                                  */
/*       [3]  all of the following:                                          */
/*         [3a]  subprocedure:                                               */
/*           [3a1]  take the money                                           */
/*           [3a2]  run                                                      */
/*         [3b]  subprocedure:                                               */
/*           [3b1]  cut the power                                            */
/*           [3b2]  cut the phone line                                       */
```

```
/*       [4]  evade the police                                              */
/* This indicates that the postcondition of step 1 will become the precondition  */
/* of step 2.  In addition, 2 must precede both 3a1 and 3b1, and both 3a2 and    */
/* 3b2 must precede 4.  With respect to the subprocedures, 3a1 must precede      */
/* 3a2, but there is no direct relationship established between 3a1 and any step */
/* in the 3b's.                                                                  */
/* The steps in the above task are: [1,2,3a1,3a2,3b1,3b2,4].  At present, one    */
/* cannot specify "3a" or "3b" as step numbers, nor will the subprocedure mech-  */
/* anisms recognize "3a" and "3b" as subprocedures as of yet.  This is a current */
/* limitation that should be an easy enough fix.                                 */
/*                                                                               */
/* Unordered Actions:                                                            */
/*   An unordered action is one that can be done at any time so long as its      */
/* preconditions are met.  Unordered actions are placed in front of the action   */
/* that depends on it and they are marked with a star:                           */
/*       [4]  do something                                                       */
/*       [5]  * do this at any time as long as it is before [6]                  */
/*       [6]  do this after [4] and after [5]                                    */
/*                                                                               */
/* Concept of Step Dependency:                                                   */
/*   The step arrangement of a task CANNOT override or veto the class-defined    */
/* operations.  Thus, if an operation has a precondition defined at the class    */
/* level, the task manipulation routines will not permit the user to do anything */
/* that would violate them.  The term "step dependency" defines a partial order  */
/* on the primitive operations which describe those actions that must precede    */
/* other actions.  For the above example, if "money is taken" is a precondition  */
/* for the class-defined operation "run" then 3a2 is dependent on 3a1.   If,     */
/* however, cutting the power is not a precondition for cutting the phone line   */
/* at the class level then 3b2 is not dependent on 3b1.  Independent actions may */
/* be shuffled around -- so a swap step command on 3b1 and 3b2 would succeed but */
/* with 3a1 and 3a2 it would fail.                                               */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's task definition data structure                                    */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/*   A task is defined as the composite of the following facts partitioned with- */
/* in the same dynamically declared module.                                      */
/*                                                                               */
/* BASIC TASK FACTS:                                                             */
/*                                                                               */
/*   task(<task name>, <actor class>, <associated props and characters>).        */
/*      The <task name> should be in the form of a verb phrase similar to an     */
/*      operation, so that it can be constructed as an operation in lessons in   */
/*      which the classes are instantiated but the task is not.  <actor class>   */
/*      is restricted to be a derived class of the standard 'character' class.   */
/*      (Those tasks which are actor independent may use 'character' as the      */
/*      actor class.  <assoc. props and characters> are the other required items */
/*      to be instantiated for this task to be available.                        */
/*      NOTE:  <actor class> is a list pair and <ass. props and char.> is a list */
/*          of pairs corresponding to (<local task prop name>, <class>).  The    */
/*          <local task prop name> is the same as <class> unless the task has    */
/*          more than one of the same <class>, in which case the subsequent      */
/*          items are augmented with numerals starting with one.                 */
/*                                                                               */
/*   initial_conditions(<task object>,  <initial conditions>).                   */
/*   objectives(<task object>, <objectives>).                                    */
/*      Maintains the initial conditions and the objectives of each object.  The */
/*      <task object> is the locally declared task instance name and <initial    */
/*      conditions> and <objectives> are lists of properties:                    */
/*      There is precisely one entry for each property set defined for the class */
```

```
/*      and components and one extra entry for each hideable property set.       */
/*         <initial conditions> == list of <property>                            */
/*         <objectives>         == list of <objective>                           */
/*         <objective>          == <property> | immaterial(<property set>)       */
/*                                                                               */
/* TASK PROCEDURE NET FACTS:                                                     */
/*                                                                               */
/*   stage(<stage name>, <outgoing split type>, <split joining stage>,           */
/*         <incoming split type>, <incoming action list>, <split stack>).        */
/*      Defines the node for a task's procedural net.  The following are the     */
/*      particulars for the arguments:                                           */
/*         <stage name>         == start | done | <Q-stage> | <J-stage>          */
/*         <Q-stage>            == q<number>                                     */
/*         <J-stage>            == join<number>                                  */
/*             Q stages are for actions out.  Join stages are for collecting and */
/*             disambiguiating the joining of actions together.                  */
/*         <outgoing split type>== no-actions | linear | or-split | and-split    */
/*         <split joining stage>== <Q-stage>                                     */
/*         <incoming split type>== linear | joining                              */
/*         <incoming action list> == <incoming action)*                          */
/*         <incoming action>    == <stage name> <action index> <split index>     */
/*         <action index>       == <action number> | lambda                      */
/*         <split index>        == <natural>                                     */
/*             <split index> will always be 1 for and-split and linear stages.   */
/*             <split index> will be 1 or greater for or-split legs.             */
/*         <split stack>        == <split stack entry>*                          */
/*         <split stack entry>  == <split stage> and|or <split index>            */
/*             The stack of open splits at a given stage.                        */
/*       The procedure net defines the existence of and-splits and or-splits.    */
/*       An or-split means that there exists more than one solution to a subtask */
/*       and the student only has to complete one of the subtasks.  An and-split */
/*       means that there exists multiple subtasks that the student must perform */
/*       all of.  These subtasks may be done in parallel, so the student can do  */
/*       any of the actions in the subtasks in any order so long as order is     */
/*       maintained among the subtasks (so given subtasks [a,b,c] and [d,e,f],   */
/*       a student may do [a,b,c,d,e,f], [d,e,f,a,b,c], [a,d,b,e,c,f], etc.)     */
/*                                                                               */
/*   action(<index #>, <operator>, <step-wise precondition list>,                */
/*        <deterministic side effect list>, <probabilistic side effect list>,    */
/*        <new message>).                                                        */
/*       Actions represent a transition in the procedural net (and the net       */
/*       manipulation routines refer to actions as "transition").  The operator  */
/*       is currently stored in cached form, but this will change as task        */
/*       instantiations for the lesson will require an instantiatable form.      */
/*                                                                               */
/*    guaranteed_state(<stage>,<list of guaranteed properties>).                 */
/*       Known as "necessary_postconditions" in previous versions of MEBuilder.  */
/*       The list of guaranteed properties consist of a list of either a single  */
/*       property or a list of multiple properties from the same property set.   */
/*       The list of multiple properties is based on cached property sets.       */
/*       guaranteed_stages are recalculated at each change in the procedural     */
/*       graph.                                                                  */
/*                                                                               */
/*   The associated props and characters will be refered to by the use of gensym */
/* on the printed name of the class when necessary to avoid ambiguity.  So a     */
/* task called repair flashlight which operates on two flashlights would be      */
/* declared as task(repair,repairman,[flashlight,flashlight]), and the task      */
/* would be instantiated as task(repair,repairman,[flashlight1,flashlight2]).    */
/*                                                                               */
/*   Each task is maintained in its own module.  In the interface, the teacher   */
/* will select the task to work on.  The library manager will maintain infor-    */
```

```
/* mation about the dependencies on the lesson, i.e. which classes are required  */
/* in order to build the lesson -- these are based solely on the prop facts.     */
/*                                                                               */
/*   Tasks are not directly accessed by outside modules but are instantiated by  */
/* lessons.                                                                      */
/*                                                                               */
/* OTHER TASK FACTS:                                                             */
/*                                                                               */
/*   relation(<relation name>, <object list>, <definition>).                     */
/*      These are very similar to that of the class definition except that these */
/*      relations do not describe the state of an object but rather a state of   */
/*      the task.  The compilation process will have to be arranged so that the  */
/*      <relation name> can in fact take an object name and cache it.  However,  */
/*      for now the relation name will be established as is.                     */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Commands provided by this module (including those not yet implemented)        */
/* Documentation on these commands can be found in the user's manual.            */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* Note on legend:  <step> is the index of an operation within the task as shown */
/*                  by the view task command (which see):                        */
/*                                                                               */
/* TASK COMMANDS EXPORTED TO OTHER LOOPS:                              LOOPS:    */
/*    create task [named <task>]                                       Main      */
/*    work on task [named <task>]                                      Main      */
/*    remove task [named <task>]                                       Main      */
/*    restore task [named <task>]                                      Main      */
/*    check task [named <task>]                                        All       */
/*    view situation [in <task>] [after <step>]                        All       */
/*    view initial conditions [in <task>] [for <object>]               All       */
/*    view objectives [in <task>] [for <object>]                       All       */
/*                                                                               */
/* TASK MANIPULATION COMMANDS:                                         LOOPS:    */
/*    wizard on | off (Undocumented commands)                          Task      */
/*       NOTE:  Sets/Releases the debugflag bit for means-ends                   */
/*    check task                                                       Task      */
/*    reset task                                                       Task      */
/*    view task                                                        Task      */
/*    set initial conditions [for <object>]                            Task      */
/*    view initial conditions [for <object>]                           Task      */
/*    set objectives [for <object>]                                    Task      */
/*    view objectives [for <object>]                                   Task      */
/*    view step dependencies                                           Task      */
/*    view situation [after <step>]                                    Task      */
/*    find splits                                                      Task      */
/*    combine step [number <step>] [with <step>]                       Task      */
/*    modify step [number <step>]                                      Task      */
/*    swap step [number <step>] [with <step>]                          Task      */
/*    move step [number <step>]                                        Task      */
/*    create summary fact                                              Task      */
/*    remove summary fact                                              Task      */
/*    view summary fact                                                Task      */
/*    modify summary fact                                              Task      */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported Predicates.  All predicates intended for external use are prefixed   */
/*  with "mebuildTSK_"                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* MAIN APPLICATION LOOP -- APPLICATION PREDICATES:                              */
```

```
/*   mebuildTSK_setup.                                                    */
/*        -- Initializes all the domains, commands, and templates for the main    */
/*           mebuild application command loop.                          */
/*   mebuildTSK_include_view_commands(+Loop).                             */
/*        -- Provides Loop with the task definition view commands for use in      */
/*           the lesson loop.                                           */
/*   mebuildTSK_task_definition_initialize.                               */
/*        -- Initializes the task_definition database.                   */
/*   mebuildTSK_task_definition_shutdown.                                 */
/*        -- Clears all task_definition facts from the Prolog database and erases */
/*           all dynamically-created modules.  Used by MEBuild's quit command.    */
/*   mebuildTSK_append_autosave.                                          */
/*        -- Appends all task definition data in memory to the autosave file.     */
/*                                                                        */
/* COMMAND INVOKED PREDICATES FOR MAIN APPLICATION LOOP:                  */
/*   mebuildTSK_define_task(+Task).                                       */
/*        -- Provides user access for creating a new task.  Invokes the task      */
/*           building application loop on the newly created task.       */
/*   mebuildTSK_remove_task(+Task).                                       */
/*        -- Marks a task as "deleted" (refer to library module).        */
/*   mebuildTSK_restore_task(+Task).                                      */
/*        -- Restores a removed task in memory (i.e. undeletes it)       */
/*   mebuildTSK_work_on_task(+Task).                                      */
/*        -- Loads in (if not loaded) and invokes the task building application   */
/*           loop on an existing task.                                  */
/*                                                                        */
/* COMMAND-INVOKED PREDICATES FROM TASK LOOP THAT ARE ALSO EXPORTED:      */
/*   mebuildTSK_test_task_integrity(+Task).                               */
/*        -- Ensures that the task is consistent with the class definitions (it   */
/*           should be OK so long as no class was updated after the task was last */
/*           saved to disk.                                             */
/*    mebuildTSK_view_initial_conditions(+Task,+Object).                  */
/*        -- Lists the Object's initial conditions as currently specified.       */
/*    mebuildTSK_view_objectives(+Task,+Object).                          */
/*        -- Lists the Object's objectives as currently specified.       */
/*    mebuildTSK_view_task(+Task).                                        */
/*        -- Prints a listing of the current solution to the problem.    */
/*    mebuildTSK_view_situation(+Task,+Step).                             */
/*        -- Prints a listing of what the situation should look like after doing  */
/*           step number Step.                                          */
/*                                                                        */
/* NON-COMMAND INVOKED EXPORTED PREDICATES:                               */
/*   mebuildTSK_get_relation(+Task,-Relation,-ClassList,-Definition)      */
/*        -- Returns a summary defined in the task.                      */
/*   mebuildTSK_cache_lesson_facts(+LessonModule,+LessonObjects,+Tasks)   */
/*        -- Prepares and caches all task-relevant data to the lesson compiler.   */
/*   mebuildTSK_exported_reset_task(+Task)                                */
/*        -- Used by mebuildLES to initialize a task upon entering the "work on   */
/*           lesson" command.                                           */
/*   mebuildTSK_assignment_of_objects(+Instance/Class Pairs,+Task,-Map)   */
/*        -- Takes a list of instances and classes, and the list of objects re-   */
/*           quired for a given task and returns a mapping of a the task-based    */
/*           ordinalized instance name (object,[second object],etc.) to the       */
/*           lesson's concrete instance name based on the cast and prop lists.    */
/*   mebuildTSK_first_task_that_employs_object(+Tasks,+Class,-Task)       */
/*   mebuildTSK_last_task_that_employs_object(+Tasks,+Class,-Task)        */
/*        -- Returns the Task among Tasks that fits the predicate name.  */
/*   mebuildTSK_get_initial_conditions_of_instance(+Task,+Instance,+Class,-IC)   */
/*        -- Retrieves the initial conditions of the +Class object in +Task and   */
/*           instantiates the initial conditions list to contain the concrete     */
/*           Instance name.                                             */
```

```
/*   mebuildTSK_get_objectives_of_instance(+Task,+Instance,+Class,-OBJ)          */
/*        -- Retrieves the objectives of the +Class object in +Task and          */
/*           instantiates the objectives list to contain the concrete Instance.  */
/*   mebuildTSK_employs_object(+Task,+Class)                                     */
/*        -- Predicate that succeeds if any object of type Class is used in Task. */
/*   mebuildTSK_get_state_of_instance(+Task,+Instance,+Class,+Stage,-State).     */
/*        -- Effectively returns the instantiated guaranteed state for the       */
/*           Instance of type Class in Task at Stage.                            */
/*   mebuildTSK_get_last_changes_in_instance(+Task,+Instance,+Class,+Stage,      */
/*                                           -State).                            */
/*        -- Returns the combination of all the last changes or the significant  */
/*           changes made in the named Instance of Class prior to named Stage in */
/*           the Task.  Used to help determine immaterial or unimportant goal    */
/*           entries.                                                            */
/*   mebuildTSK_export_task(+Task,+File).                                        */
/*        -- Send out the task definition in memory to the file.                 */
/*                                                                               */
/*********************************************************************************/
```

TAB 4. MEBUILDER LESSON MODULE (MEBuildLES)

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1            (MEBUILDER)    */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Lesson Definition Module -- Version 1.01                        */
/*                                                                           */
/*   This module defines what a lesson is as a structured instantiation of tasks */
/* objects to produce a concrete lesson entity runnable in the means-ends tu- */
/* toring shell.  It also provides a quick and effective means of building be- */
/* ginner level through advanced level problems in the space.                */
/*   Lessons consist of one or more problems, which contain different scenarios */
/* and provide different parameters to some of the probabilistic events in the */
/* instantiated tasks.  The intent is that the problems should be ordered in in- */
/* creasing level of difficulty or in accordance with an accepted curricular or- */
/* dering.                                                                   */
/*                                                                           */
/* Version History:                                                          */
/*   1.0    -- Version released for use in the experiment of Summer Quarter 94. */
/*   1.01   -- Version 1.0 fully documented.                                 */
/*                                                                           */
/*                                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's lesson fundamentals                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                           */
/*   A lesson is a workbook of exercises for the student to perform.  The exer- */
/* cises should be used to present different problems to student, in general  */
/* they should be increasing in difficulty and/or complexity.  Examples of the */
/* different types of problems that would be suggested are given below.  These */
/* types could certainly be combined in a given workbook.                    */
/*     A.  Increasing Complexity                                             */
/*         Prob 1 = basic task with all negative pitfalls (i.e. all prob. side */
/*                  effects turned off)                                      */
/*         Probs 2-n = same task with the negative pitfalls given increasing */
/*                  likelihood and/or adversarial agents working faster.     */
/*     B.  Walk, Crawl, Run                                                  */
/*         Prob 1 = first chunk of task                                      */
/*         Prob 2 - (n-1) = next chunks of task                              */
/*         Prob n = whole task (comprehensive fashion)                       */
/*     C.  Different Equipment                                               */
/*         Prob 1 = task on a basic prop (i.e. car)                          */
/*         Probs 2-n = task on more specific types of prop (i.e. domestic cars, */
/*                  foreign cars, etc. which might have particular needs or  */
/*                  considerations)                                          */
/*     D.  Different Roles                                                   */
/*         Probs 1-n = If a given task has n team members cooperating together, */
/*                  then for each problem give the student a different role to */
/*                  play.  Either prob. 1 or n should be the guy in charge.  */
/*     E.  Different Tasks                                                   */
/*         Probs 1-n = If the library contains n different tasks involving the */
/*                  same actor class and same sets of props and all n tasks in- */
/*                  volve a general subject matter, then have the student do the */
/*                  n tasks, one at a time.  (Ex. for a mechanic and a car    */
/*                  engine, one could have the student replace the water pump */
/*                  gasket in problem 1, fix a fuel injector in problem 2, ...) */
/*                                                                           */
/*   During construction of the tasks, the teacher talks in terms of abstract */
/* objects (car, character, etc.).  In a lesson, the teacher now talks in terms */
```

```
/* of concrete objects (proper nouns -> Jim, Jim's car, etc.).  Lessons contain  */
/* a "cast" and a set of "props", and setting up the lesson is done in terms of  */
/* naming a "setting" or a "scene" and having the student perform a "role" in    */
/* the scene.  The task could be viewed as a script, indicating how the actors   */
/* behave under different stimuli.                                               */
/*   The lesson itself has an introductory text along with the cast and prop     */
/* listing.  Each problem consists of an introductory text, along with the des-  */
/* criptions of the particular scene and goal, along with options that allow for */
/* different behaviours among problems.                                          */
/*   The intent of the MEBuilder system is to have the teacher spend the major-  */
/* ity of the time here, at the highest level of the MEBuilder hierarchy -- do-  */
/* ing choreographic and situational things rather than digging in the weeds of  */
/* the task and object layers.  The library management system is there so a huge */
/* reusable object library could be created  of both objects and tasks.  So the  */
/* lesson interface needs to be the most robust.                                 */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's lesson definition data structure                                  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/*   A lesson is defined as the composite of the following facts partitioned     */
/* within the same dynamically declared module.  When a lesson is compiled into  */
/* means-ends form, this data structure will help keep repetition to a minimum.  */
/*   The associated class definition information is copied in to assist, however */
/* the tasks are not copied into the dynamic module.  Their information is kept  */
/* singly instantiated in its own module.                                        */
/*                                                                               */
/* BASIC LESSON FACTS:                                                           */
/*                                                                               */
/*   lesson(<lesson name>, <cast>, <props>, <tasks>).                            */
/*     The <lesson name> is strictly unique to the environment.  The cast must   */
/*     be non-empty.  The list of tasks are a subset of all those which could be */
/*     instantiated from the <cast> and <props>.  The <cast> and <props> are     */
/*     2-tuples of (<instance name> <class name>) or more appropriately, (<role  */
/*     name/prop name> <class>).  Throughout this module, cast members will be   */
/*     referred by their role, and props by their prop name.                     */
/*                                                                               */
/*   lesson_intro(<text>).                                                       */
/*     Global introduction for the lesson.                                       */
/*                                                                               */
/* PROBLEM FACTS:                                                                */
/*                                                                               */
/*   problem(<number>, <name>, <student's role>, <cast>, <props>).               */
/*     The problem fact provides the basic information for the problem.  The     */
/*     cast and prop arguments allow the teacher to specify that a cast member   */
/*     or prop is a derived class of that given in the lesson level.  Generally, */
/*     the cast members and prop lists match that of the lesson.                 */
/*     NOTE:  Currently there is no hook in place for problem options.  The      */
/*     side_effect_override below is sort of a hook but it's not used.  They     */
/*     could be placed as a sixth argument here or they could be implemented as  */
/*     separate facts (in fact, the cast and props could be treated in override  */
/*     fashion as well).                                                         */
/*                                                                               */
/*   initial_setting(<number>, <initial condition list>).                        */
/*     Analogous to the initial_conditions of each task, except that the initial */
/*     setting is a composite of all the initial_conditions of each prop.        */
/*                                                                               */
/*   objectives(<number>, <cast member>, <objective list>).                      */
/*     Unlike the initial setting, the objectives are broken out by cast member  */
/*     since each cast member will act independently based on the scene.  These  */
/*     will be used to derive the agent facts in the metutor lesson.  Also note  */
```

```
/*     that the objectives per cast member are fixed for all problems in the     */
/*     lesson.  This is the global "resting state" for the cast member.          */
/*                                                                               */
/*   problem_intro(<number>, <text>).                                            */
/*     Introductory text for the problem.                                        */
/*                                                                               */
/*   side_effect_override(<number>,<op>,<context>,<change>,<prob>).              */
/*     NOT IMPLEMENTED.  This was the first swag at a problem option concept.    */
/*     It may be sufficient for the event-specific probabilities, but it is      */
/*     probably better to have a more global options construct.                  */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Commands provided by this module (including those not yet implemented)        */
/* Documentation on these commands can be found in the user's manual.            */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* LESSON COMMAND EXPORTED TO OTHER LOOPS:                          LOOPS:       */
/*   create lesson [named <lesson>]                                 Main         */
/*   work on lesson [named <lesson>]                                Main         */
/*   remove lesson [named <lesson>]                                 Main         */
/*   restore lesson [named <lesson>]                                Main         */
/*   check lesson [named <lesson>]                                  All          */
/*   view lesson [named <lesson>]                                   All          */
/*   compile lesson [named <lesson>]                                All          */
/*   run lesson [named <lesson>]                                    All          */
/*                                                                               */
/* The Lesson Loop is invoked through the create lesson and work on lesson cmds. */
/* LESSON MANIPULATION COMMANDS:                                    LOOPS:       */
/*   wizard on | off (Undocumented commands)                        Lesson       */
/*      NOTE:  Sets/Releases the debugflag bit for means-ends                    */
/*   check lesson                                                   Lesson       */
/*   view lesson                                                    Lesson       */
/*   compile lesson                                                 Lesson       */
/*   run lesson                                                     Lesson       */
/*   edit lesson intro                                              Lesson       */
/*   view lesson intro                                              Lesson       */
/*   create problem [named <problem>]                               Lesson       */
/*   work on problem [number <problem no>]                          Lesson       */
/*   work on problem [named <problem>]                              Lesson       */
/*   order problems                                                 Lesson       */
/*   remove problem [number <problem no>]                           Lesson       */
/*   remove problem [named <problem>]                               Lesson       */
/*   check problem [number <problem no>]                            Lesson       */
/*   check problem [named <problem>]                                Lesson       */
/*   view problem [number <problem no>]                             Lesson       */
/*   view problem [named <problem>]                                 Lesson       */
/*                                                                               */
/* The Problem Loop is invoked via the create problem and work on problem cmds   */
/* PROBLEM MANIPULATION COMMANDS:                                   LOOPS:       */
/*   wizard on | off (Undocumented commands)                        Problem      */
/*      NOTE:  Sets/Releases the debugflag bit for means-ends                    */
/*   check problem                                                  Problem      */
/*   view problem                                                   Problem      */
/*   edit problem intro                                             Problem      */
/*   view problem intro                                             Problem      */
/*   set scene [for <cast member or prop>]                          Problem      */
/*   view scene [for <cast member or prop>]                         Problem      */
/*   set goal [for <cast member or prop>]                           Problem      */
/*   view goal [for <cast member or prop>]                          Problem      */
/*   set options                                                    Problem      */
/*                                                                               */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported Predicates.  All predicates intended for external use are prefixed   */
/*  with "mebuildLES_" except for those from the lesson compiler "mebuildCMP_"   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                                */
/* MAIN APPLICATION LOOP -- APPLICATION PREDICATES:                               */
/*   mebuildLES_setup.                                                            */
/*       -- Initializes all the domains, commands, and templates for the main     */
/*          mebuild application command loop.                                     */
/*   mebuildLES_lesson_definition_initialize.                                     */
/*       -- Initializes the lesson_definition database.                           */
/*   mebuildLES_lesson_definition_shutdown.                                       */
/*       -- Clears all lesson_definition facts from the Prolog database, erasing  */
/*          all dynamically-created modules.  Used by MEBuild's quit command.     */
/*   mebuildLES_append_autosave.                                                  */
/*       -- Appends all lesson definition data in memory to the autosave file.    */
/*                                                                                */
/* COMMAND-INVOKED PREDICATES FOR MAIN APPLICATION LOOP:                          */
/*   mebuildLES_define_lesson(+Lesson).                                           */
/*       -- Provides user access for creating a new task.  Invokes the task       */
/*          building application loop on the newly created task.                  */
/*   mebuildLES_remove_lesson(+Lesson).                                           */
/*       -- Marks a lesson as "deleted" (refer to library module).                */
/*   mebuildLES_restore_lesson(+Lesson).                                          */
/*       -- Restores a removed lesson in memory (i.e. undeletes it)               */
/*   mebuildLES_work_on_lesson(+Lesson).                                          */
/*       -- Loads in (if not loaded) and invokes the lesson building application  */
/*          loop on an existing lesson.                                           */
/*                                                                                */
/* COMMAND-INVOKED PREDICATES FROM LESSON LOOP THAT ARE ALSO EXPORTED:            */
/*   mebuildLES_check_lesson(+Lesson).                                            */
/*       -- Performs integrity checks on all cast members, props, tasks and       */
/*          problems to ensure that the lesson could be run successfully.         */
/*   mebuildLES_view_lesson(+Lesson).                                             */
/*       -- Prints out a table of all the primary lesson data.                    */
/*   mebuildCMP_compile_lesson(+Lesson).                                          */
/*       -- Compiles the lesson in METutor-readable form and saves to a .pl file  */
/*          on disk.                                                              */
/*   mebuildLES_run_lesson(+Lesson).                                              */
/*       -- Invokes the METutor shell in order to run a lesson.                   */
/*                                                                                */
/* NON-COMMAND INVOKED EXPORTED PREDICATES:                                       */
/*   mebuildLES_export_lesson(+Lesson,+File).                                     */
/*       -- Send out the lesson definition in memory to the file.                 */
/*                                                                                */
/**********************************************************************************/
```

TAB 5. MEBUILDER LESSON COMPILER (MEBuildCMP)

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1            (MEBUILDER)    */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Lesson Compiler -- Version 1.01                                 */
/*                                                                           */
/*    The MEBuilder lesson compiler takes an MEBuilder-constructed lesson and */
/* creates an means-ends conversion of it into an analogous database partition. */
/* The database partition is named the same as the lesson itself except that it */
/* contains the added ending "_me".  When a lesson is run, the partition sent */
/* to the metutor_shell is the _me partition.                                */
/*    NOTE:  Need to extract all lesson caching stuff out of the mebuild_class */
/* definition and get it in here in order to achieve some degree of consistency. */
/*                                                                           */
/* Version History:                                                          */
/*   1.0       -- Version released for use in the Summer Qtr 94 experiment.  */
/*   1.01      -- Fully commented                                            */
/*                                                                           */
/*                                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Description of the Compilation Process                                    */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                           */
/*    "Compilation" is probably a less accurate description of what happens here */
/* than "translation".  Basically, the lesson representation constructed in  */
/* MEBuilder form is translated into a form that METutor can use.  That is, the */
/* procedural net and class definitions are converted into a sequence of recom- */
/* mended, precondition, addpostcondition, and deletepostcondition clauses --  */
/* along with all the other clauses that METutor uses as support -- such as the */
/* randchange, singular, plural, etc.                                        */
/*                                                                           */
/*    The lesson compiler will then attempt to solve the problem in as many ways */
/* as feasible in order to detect any mismatching of the instantiated tasks.  */
/* The algorithm for doing this is described below.                          */
/*                                                                           */
/*    The following is the sequence of actions that the lesson compiler does: */
/*                                                                           */
/* (1)  Class integrity.  (Performed by mebuild_class_definition)            */
/*      (a)  Definition Integrity.  All property definitions, operators,     */
/*           components, etc. must be completely and consistently defined.   */
/*           Checks for errors which may occur when an operator is defined and */
/*           then the property set it is defined on is deleted.  Failure is  */
/*           fatal and compilation stops.                                    */
/*      (b)  Definition Completeness.  All properties are tested to ensure they */
/*           have an associated operation (however, of course, some properties */
/*           will not have operators which make them occur).  Failure only   */
/*           produces warnings and compilation continues.                    */
/*                                                                           */
/* (2)  Task Integrity.  (Performed by mebuild_task_definition)              */
/*      (a)  Procedure Graph Integrity.  The procedure net must be closed with */
/*           respect to the "start" and "done" stages, meaning that there must */
/*           be no stages with no outgoing actions and all must reach "done". */
/*           All error recovery procedures must reach "return".  Any severed */
/*           portions of the procedure graph must be completely severed and  */
/*           will only produce a warning message.  All other procedure net   */
/*           integrity violations are fatal and compilation stops.           */
/*      (b)  Action Integrity.  The various actions in the net are then tested */
/*           to insure that all operators, properties, etc. are defined by the */
```

```
/*          class definition.  Failure is fatal and compilation stops.          */
/*      (c) Split Integrity.  AND-splits must have legs which are mutually      */
/*          exclusive in activity (however, Type III preconditions are still OK */
/*          for forcing a partial ordering of AND-split actions).  OR-split     */
/*          initial actions must all act on the same property set and have an   */
/*          entry precondition which is an unused member of the property set.   */
/*          Failure is fatal and compilation stops.                             */
/*      (d) Semantic Integrity.  The guaranteed_state at "done" must contain    */
/*          the objectives without possibility of other states.                 */
/*                                                                              */
/* (3)  Lesson Integrity.  (Performed by mebuild_lesson_definition)             */
/*      (a) Object Integrity.  Insures that the cast, props, and tasks are      */
/*          consistent as declared for the lesson.                              */
/*      (b) Objective Integrity.  Insures that the objectives listed exist for  */
/*          the cast and prop members.                                          */
/*      (c) Problem Integrity.  Insures that the initial settings and the over- */
/*          rides for each problem are well-defined and consistent.             */
/*                                                                              */
/* (4)  Translation.  (performed here)                                          */
/*      The specific algorithms for the translation of the means-ends facts are */
/*      described in the translation section of this file.                      */
/*                                                                              */
/* (5)  Lesson Testing.  (performed here)                                       */
/*      The individual tasks are tested for semantic and traversal integrity.   */
/*      Not sure yet what it will mean to traverse a full forest of procedure   */
/*      graphs.                                                                 */
/*                                                                              */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's Compiled Lesson data structure                                   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                              */
/*    The compilation data structure for MEBuilder-based lessons use a templated */
/* form of the means-ends search space -- i.e. it uses macros.  Future implem-  */
/* entations of MEBuilder might find it better not to use macros -- but the     */
/* advantage of using macros is that large volumes of lesson material can be    */
/* stored in less space.  The disadvantage is that macro expansion is done at   */
/* run time which is slower.                                                    */
/*                                                                              */
/*    Some of these macros, fortunately, can be directly determined from the    */
/* class definition module -- requiring no additional translation for effect.   */
/* These are the hideable_facts, summary_facts, singular, and plural designators */
/* for components, the daemon-based randchange facts, and the class-defined     */
/* display data information for properties and operations.  (NOTE:  Currently,  */
/* not all these will work properly since, for example, summary facts are single */
/* object items at this writing and are not included for tasks and lessons.     */
/* Summary facts are also recursive by design but not implemented as such yet.  */
/* These are shortfalls being worked on at this moment.                         */
/*                                                                              */
/*    The following are the compiled lesson facts for this version.  The macro  */
/* expansion done at runtime produces instantiated facts which do not contain   */
/* the added "_t" ending.  Those without a "_t" ending are not macro expanded.  */
/*                                                                              */
/*    The template forms are one of two types and are based on class domains:   */
/*        +Template              ::= <class>*                                   */
/*        +QuantifiedTemplate    ::= (some(<class>),forall(<class>))*           */
/*    The members of the template are referenced in the body of the macro as    */
/*    arg(X) or parg(X) for the object's name and possessive forms (if followed */
/*    by the component name for properties and operations on components).       */
/*    Macro-based arguments for properties, operations, etc. will be listed     */
/*    as MacroPropertyList, MacroOperation, etc.                                */
/*    NOTE:  For cases when there is exactly one element in the domain for all  */
```

```
/*          problems in the lesson, the lesson compiler will perform partial    */
/*          macro expansion (still preserve the "_t" macro form but reduce the  */
/*          macro-arguments to expand).                                         */
/*    NOTE:  Need to explore the agent fact to see if it really is needed now.  */
/*                                                                              */
/*    lesson(<name>).                                                           */
/*    lesson_intro(<text>).                                                     */
/*        Copied from the lesson definition.                                    */
/*                                                                              */
/*    goal_t(<quantified template>, <macro property list>).                     */
/*        Serves as the single goal list.  If any template elements are quanti- */
/*        field existentially, then multiple goal facts will be expanded and    */
/*        satisfying any one will be sufficient for achieving the goal.         */
/*        There is only one goal_t per lesson.                                  */
/*                                                                              */
/*    problem(<problem number>, <problem name>, <student role>).                */
/*        Partial facts copied from the lesson definition.                      */
/*                                                                              */
/*    problem_domain(<problem number>, <class>, <instances>).                   */
/*        Used to supply the names of the objects of like type in the arg(X) and */
/*        arg(X) facts.  It is an expanded copy of the reduced cast and re-     */
/*        duced props arguments in the lesson's problem fact.                   */
/*                                                                              */
/*    problem_intro(<problem number>, <text>).                                  */
/*        Also copied from the lesson definition.                               */
/*                                                                              */
/*    start_state_t(<problem#>, <quantified template>, <macro property list>).  */
/*        Start_state_t is a macro which currently will always have a null tem- */
/*        plate and a fully expanded <macro property list>.                     */
/*                                                                              */
/*    singular_t(<template>, <macro object>).                                   */
/*    plural_t(<template>, <macro object>).                                     */
/*        For component's, <template> will be a single class entry.  Full cast  */
/*        members and props who use singular and plural facts will use a null   */
/*        template and a fully expanded name in the macro object argument.      */
/*        NOTE:  These macros are fully class defined and are produced by the   */
/*               mebuild_class_definition module.                               */
/*                                                                              */
/*    recommended_t(<role>, <template>, <macro property list>,                  */
/*            <macro operation>).                                               */
/*    precondition_t(<role>, <template>, <macro operation>, <macro context>,    */
/*            <macro precondition list>).                                       */
/*    addpostcondition_t(<role>, <template>, <macro operation>, <macro context>, */
/*            <macro addpostcondition list>).                                   */
/*    deletepostcondition_t(<role>, <template>, <macro operation>,              */
/*            <macro deletepostcondition list>).                                */
/*        The templated forms of the means-ends facts.  The <role> argument is  */
/*        optional, but would be produced whenever the rule applies to a cast   */
/*        member other than the student's role.                                 */
/*                                                                              */
/*    randchange_t(<template>, <macro application>, <macro context>,            */
/*            <macro delete list>, <macro add list>,  <macro count/prob>,       */
/*            <macro message>).                                                 */
/*        Describes random events that occur -- or those events that occur in-  */
/*        dependent of a user's action.  <macro application> includes init,     */
/*        corresponding to random changes at the start of the problem or any    */
/*        templated operation.  <macro count/prob> corresponds to the use of    */
/*        counters or probabilities to indicate activation of the randchange.   */
/*        (Currently it is only <macro prob> which is a raw probability value.  */
/*        count macros are not yet implemented.)  <macro delete> and <macro add> */
/*        are macro property lists and macro message is a message template for  */
```

```
/*        warning output to the student.                                    */
/*                                                                          */
/*    hideable_facts_t(<quantified template>, <macro property set name>,    */
/*             <macro property set members>).                               */
/*        Allows the METutor shell to identify what information is to be with-*/
/*        held from the user when the unknown(<expanded property set name>) fact */
/*        is true in the current state.                                     */
/*        NOTE:  These macros are fully class defined and are produced by the */
/*               mebuild_class_definition module.                           */
/*                                                                          */
/*    summary_fact_t(<template>, <macro summary fact>, <macro definition>). */
/*        Provides the relation definitions for the lesson.                 */
/*                                                                          */
/*    apply_text_t(<template>, <macro operation>, <macro context>, <macro msg>). */
/*        Provide messages that will override the default apply text information */
/*        generated by the METutor shell.                                   */
/*                                                                          */
/* IMPORTANT NOTE:  The original intent was that the templates would be mini- */
/*    mized in order to take advantage of things like single-object domains. */
/*    This concept was abandoned because there were significant problems in */
/*    METutor's macro expansion process when the single-object domains happened */
/*    to be the student's role and the fact would up with a null template.  The */
/*    result was that some facts were not being placed in first person form, */
/*    hence causing unsolvable problems.  Therefore, *all* nouns are listed in */
/*    macro as types and no streamlining of templates is done.              */
/*                                                                          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported Predicates.  All predicates intended for external use are prefixed */
/*  with "mebuildCMP_"                                                      */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                          */
/* APPLICATION PREDICATES:                                                  */
/*    mebuildCMP_setup.                                                     */
/*        -- Initializes all the domains, commands, and templates for the main */
/*           mebuild application command loop.  (should only be templates). */
/*    mebuildCMP_compiler_initialize.              (Main App Init Routine)  */
/*        -- Initializes the compiled_lesson database.                      */
/*    mebuildCMP_compiler_shutdown.                (Main App Done Routine)  */
/*        -- Clears all compiled definition facts from the Prolog database and */
/*           erases all dynamically created modules.  Used by MEBuild's quit. */
/*    mebuildCMP_append_autosave.                  (Main App Loop Routine)  */
/*        -- Appends all compiled lessons in memory to the autosave file.   */
/*                                                                          */
/* COMPILED LESSON MANAGEMENT:                                              */
/*    mebuildCMP_compile_lesson(+Lesson).                                   */
/*        -- Top-level lesson compiling function.  Succeeds if compilation per- */
/*           forms to completion with CompiledMod containing the compiled form. */
/*           If failed, compile_lesson will fail and the data in the unreturned */
/*           CompiledMod is undefined.                                      */
/*    mebuildCMP_save_compiled_lesson(+Lesson,+File).                       */
/*        -- Ships the Compiled form of Lesson into the file File.  File is then */
/*           a fully METutor-runnable lesson.                               */
/*                                                                          */
/****************************************************************************/
```

TAB 6. MEBUILDER LIBRARY MANAGER (MEBuildLIB)

```
/*****************************************************************************/
/* Means-Ends Lesson Building Program -- Version 1              (MEBUILDER)  */
/* CPT Thomas P. Galvin, U.S. Army, Naval Postgraduate School, Monterey CA 93940 */
/*****************************************************************************/
/* MEBUILDER Library Management Module -- Version 1.02  (mebuildLIB)         */
/*                                                                           */
/* Version Updates                                                           */
/*   1.0   -- Fundamental add, load, and save operations built.              */
/*   1.01  -- Compiled file management added and queries for unsaved changed */
/*            added.                                                         */
/*   1.02  -- Added procedural stubs for delete and undelete commands, a purge */
/*            library command, and the link management commands (add link,   */
/*            remove link, plus others).  The purpose of the stubs is to allow */
/*            easier inclusion of these commands in future.                  */
/*            Code has been thoroughly commented to describe decisions made and */
/*            requirements for future implementation.                        */
/*                                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* MEBuilder's Library data structure                                        */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                           */
/*    The local library is stored in ./lib from the teacher's working directory */
/* The directory contains a special file called "mebuild.lib" which stored the */
/* library information.  The local library is prolog_file_type marked with   */
/* mebuild_local_library_definition_file.  The following is the library data */
/* structure:                                                                */
/*                                                                           */
/*    library_class_entry(+Class,+FileName,+DateTime-Stamp,+ClassDependencies). */
/*    library_task_entry(+Task,+FileName,+DateTime-Stamp, +ClassDependencies). */
/*    library_lesson_entry(+Lesson,+FileName,+DateTime-Stamp,+ClassDependencies, */
/*                  +TaskDependencies).                                      */
/*       -- FileName is stored as an absolute file name, to prevent problems */
/*          when running MEBuilder from other directories.  If FileName is the */
/*          keyword "none", then this object has been created during the current */
/*          session only.                                                    */
/*       -- DateTime-Stamp provides the system time that the class was last  */
/*          updated using Quintus Prolog's date library.  This helps identify */
/*          when other classes or lessons need to be checked to ensure the   */
/*          versioning data is OK.  The keyword "none" is same as for FileName. */
/*       -- Dependencies are lists of classes that any entry depends on.  These */
/*          other entries, if not updated in the current database, will be   */
/*          called in automatically.                                         */
/*    library_link(+LibraryName,+RemoteLibraryDirectory)                     */
/*       -- Allows a teacher to access another local library, as long as it's */
/*          file protection is read.  Classes imported from linked libraries */
/*          will be read-only.                                               */
/*       -- Libraries will have a library name tag that will be used by the  */
/*          interface rather than the directory name.                        */
/*                                                                           */
/*    class_in_database(+Class,clean|dirty,+NewClassDep).                    */
/*    task_in_database(+Task,clean|dirty,+NewClassDep).                      */
/*    lesson_in_database(+Lesson,clean|dirty,+NewClassDep,+NewTaskDep).      */
/*    compilation_in_database(+Lesson,current|not_current).                  */
/*       -- Indicates the item has been modified and has the given new depen- */
/*          dency lists.  Compilation status is either current or not_current. */
/*                                                                           */
/*    For the purposes of using the module reservation system in utility (which */
/* is new for this version -- the tenant name is always class(<class name>),  */
```

```
/* task(<task name>), lesson(<lesson name>), or complesson(<lesson name>).       */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Commands provided by this module (including those not yet implemented)       */
/* Documentation on these commands can be found in the user's manual.            */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* LIBRARY MANAGEMENT COMMANDS:                                 LOOPS:           */
/*    create library                                            Main             */
/*    view library                                              All              */
/*    view remote library [named <library>]                     All              */
/*    link library [named <library>]                            Main             */
/*    unlink library [named <library>]                          Main             */
/*    purge library                                             Main             */
/*                                                                               */
/* CLASS ENTRY MANAGEMENT COMMANDS:                                              */
/*    load object [named <class>]                               Main             */
/*    save object [named <class>]                               Main             */
/*                                                                               */
/* TASK ENTRY MANAGEMENT COMMANDS:                                               */
/*    load task [named <task>]                                  Main             */
/*    save task                                                 Task             */
/*    save task [named <task>]                                  Main             */
/*                                                                               */
/* LESSON ENTRY MANAGEMENT COMMANDS:                                             */
/*    load lesson [named <lesson>]                              Main             */
/*    save lesson                                               Lesson           */
/*    save lesson [named <lesson>]                              Main             */
/*    save compiled lesson                                      Lesson           */
/*    save compiled lesson [named <lesson>]                     Main             */
/*                                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported Predicates.  All predicates intended for external use are prefixed   */
/*  with "mebuildLIB_"                                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                               */
/* APPLICATION PREDICATES:                                                       */
/*    mebuildLIB_setup.                                                          */
/*       -- Initializes all the domains, commands, and templates for the main    */
/*          mebuild application / class building loop                            */
/*    mebuildLIB_include_task_commands(+LoopName,+TaskName)                      */
/*       -- Makes available those commands which are to be allowed in the task   */
/*          building command loop                                                */
/*    mebuildLIB_include_lesson_commands(+LoopName,+LessonName)                  */
/*       -- Makes available those commands which are to be allowed in the lesson */
/*          building command loop                                                */
/*    mebuildLIB_library_initialize.   (Main App Init Predicate)                 */
/*       -- Initializes the library database to be empty.  Calls shutdown, and   */
/*          then loads in the system and local library files.                    */
/*    mebuildLIB_library_shutdown.     (Main App Done Predicate)                 */
/*       -- Cleans the library information from the Prolog database.  Acts upon  */
/*          any dirty items in the library interactively.                        */
/*    mebuildLIB_append_autosave(+AutosaveFile)                                  */
/*                                     (Main App Loop Predicate)                 */
/*       -- Appends all library-based data items into the autosave file.         */
/*                                                                               */
/* LIBRARY MANAGEMENT (ALL COMMAND-INVOKED):                                     */
/*    mebuildLIB_create_library.                                                 */
/*       -- Creates a new library directory in the current working directory and */
/*          initializes the mebuild.lib file.                                    */
/*    mebuildLIB_view_library.                                                   */
```

```
/*       -- Prints out a listing of the local library to the user.              */
/*    mebuildLIB_view_remote_library(+RemoteLibrary).                          */
/*       -- Prints out the listing of the given remote library.                */
/*    mebuildLIB_link_library(+RemoteLibrary)                                  */
/*       -- Would add RemoteLibrary to the link list.                          */
/*    mebuildLIB_unlink_library(+RemoteLibrary)                                */
/*       -- Would remove RemoteLibrary from the link list.                     */
/*                                                                             */
/* LIBRARY CLASS ENTRY MANAGEMENT:                                             */
/*    (COMMAND INVOKED):                                                       */
/*    mebuildLIB_load_class(+Class)                                            */
/*       -- Imports a class definition file from the library, including all    */
/*          classes in the dependencies list.                                  */
/*    mebuildLIB_save_class(+Class)                                            */
/*       -- Save the class to disk and updates the library file.               */
/*    (EXPORTED TO CLASS DEFINITION MODULE AND/OR USED HERE):                  */
/*    mebuildLIB_is_a_locally_defined_class(+Class)                            */
/*       -- Succeeds if ClassName exists as a class in the local library.      */
/*    mebuildLIB_create_library_class_entry(+Class)                            */
/*       -- Creates an entry for Class in the library.                         */
/*    mebuildLIB_delete_library_class_entry(+Class)                            */
/*       -- Marks a class as deleted.                                          */
/*    mebuildLIB_undelete_library_class_entry(+Class)                          */
/*       -- Unmarks a deleted class.                                           */
/*    mebuildLIB_purge_library_class_entry(+Class)                             */
/*       -- Permanently removed the class from the library.                    */
/*    mebuildLIB_check_load_class(+Class)                                      */
/*       -- Imports a class definition only if it isn't already loaded.        */
/*    mebuildLIB_get_loaded_classes(-Classes)                                  */
/*       -- Returns a list of all classes currently loaded in memory.         */
/*    mebuildLIB_check_save_class(+Class)                                      */
/*       -- Saves a class definition only if it is modified in memory.         */
/*    mebuildLIB_view_library_class_entry(+Class)                              */
/*       -- Pretty prints the class' entry regarding its file location, date   */
/*          time stamp of last save, and its dependency information.           */
/*    mebuildLIB_mark_library_class(+Class)                                    */
/*       -- Sets the dirty flag on the class' entry so that initialize and     */
/*          shutdown can ensure the user has a chance to save changes.         */
/*    mebuildLIB_set_library_class_dependency(+Class,+NewDependent)            */
/*    mebuildLIB_remove_library_class_dependency(+Class,+OldDependent)         */
/*       -- Sets and removes a class from the dependency list of another class. */
/*                                                                             */
/* LIBRARY TASK ENTRY MANAGEMENT:                                              */
/*    (COMMAND INVOKED):                                                       */
/*    mebuildLIB_load_task(+Task)                                              */
/*       -- Imports a task definition file from the library, including all     */
/*          classes and tasks in the dependencies list.                        */
/*    mebuildLIB_save_task(+Task)                                              */
/*       -- Save the task to disk and updates the library file.                */
/*    (EXPORTED TO CLASS DEFINITION MODULE AND/OR USED HERE):                  */
/*    mebuildLIB_is_a_locally_defined_task(+Task)                              */
/*       -- Succeeds is TaskName exists as a task in the local library.        */
/*    mebuildLIB_create_library_task_entry(+Task)                              */
/*       -- Creates an entry for Task in the library.                          */
/*    mebuildLIB_delete_library_task_entry(+Task)                              */
/*       -- Marks a task as deleted.                                           */
/*    mebuildLIB_undelete_library_task_entry(+Task)                            */
/*       -- Unmarks a deleted task.                                            */
/*    mebuildLIB_purge_library_task_entry(+Task)                               */
/*       -- Permanently removed the task from the library.                     */
/*    mebuildLIB_check_load_task(+Task)                                        */
```

```
/*       -- Imports a task definition only if it isn't already loaded.          */
/*    mebuildLIB_get_loaded_tasks(-Tasks)                                       */
/*       -- Returns a list of all tasks currently loaded in memory.            */
/*    mebuildLIB_check_save_task(+Task)                                         */
/*       -- Exports a task definition only if it has been modified.            */
/*    mebuildLIB_query_save_working_task(+Task)                                 */
/*       -- If the task is "dirty", then it queries the user if it is to be    */
/*          saved to disk before proceeding.                                   */
/*    mebuildLIB_view_library_task_entry(+Task)                                 */
/*       -- Pretty prints library task information regarding its file location, */
/*          its data time stamp, and its dependency information.               */
/*    mebuildLIB_mark_library_task(+Task)                                       */
/*       -- Sets the dirty flag on the task's entry so that initialize         */
/*          and shutdown can ensure the user has a chance to save changes.     */
/*    mebuildLIB_task_is_outdated(+Task)                                        */
/*       -- Performs a date check on all dependent entities to ensure that the */
/*          task is based on the most up to date information.  If a task or    */
/*          class has been updated since the last save of the task, then the   */
/*          task is considered untrustworthy.                                  */
/*    mebuildLIB_set_library_task_dependency(+Task,+NewDependent)               */
/*    mebuildLIB_remove_library_task_dependency(+Task,+OldDependent)            */
/*       -- Sets and removes a class or task from the dependency list of the   */
/*          task.                                                              */
/*                                                                             */
/* LIBRARY LESSON ENTRY MANAGEMENT:                                            */
/*    (COMMAND INVOKED):                                                       */
/*    mebuildLIB_load_lesson(+Lesson)                                          */
/*       -- Imports a lesson definition file from the library, including all   */
/*          classes and tasks in the dependencies list.                        */
/*    mebuildLIB_save_lesson(+Lesson)                                          */
/*       -- Save the lesson to disk and updates the library file.              */
/*    (EXPORTED TO CLASS DEFINITION MODULE AND/OR USED HERE):                  */
/*    mebuildLIB_is_a_locally_defined_lesson(+Lesson)                          */
/*       -- Succeeds is LessonName exists as a lesson in the local library.    */
/*    mebuildLIB_create_library_lesson_entry(+Lesson)                          */
/*       -- Creates an entry for Lesson in the library.                        */
/*    mebuildLIB_delete_library_lesson_entry(+Lesson)                          */
/*       -- Marks a lesson as deleted.                                         */
/*    mebuildLIB_undelete_library_lesson_entry(+Lesson)                        */
/*       -- Unmarks a deleted lesson.                                          */
/*    mebuildLIB_purge_library_lesson_entry(+Lesson)                           */
/*       -- Permanently removed the lesson from the library.                   */
/*    mebuildLIB_check_load_lesson(+Lesson)                                    */
/*       -- Imports a lesson definition only if it isn't already loaded.       */
/*    mebuildLIB_get_loaded_lessons(-Lessons)                                  */
/*       -- Returns a list of all lessons currently loaded in memory.          */
/*    mebuildLIB_query_save_working_lesson(+Lesson)                            */
/*       -- If the lesson is "dirty", then it queries the user if it is to be  */
/*          saved to disk before proceeding.                                   */
/*    mebuildLIB_view_library_lesson_entry(+Lesson)                            */
/*       -- Pretty prints library lesson information regarding its file location,*/
/*          its data time stamp, and its dependency information.               */
/*    mebuildLIB_mark_library_lesson(+Lesson)                                  */
/*       -- Sets the dirty flag on the lesson's entry so that initialize       */
/*          and shutdown can ensure the user has a chance to save changes.     */
/*    mebuildLIB_mark_compilation(+Lesson)                                     */
/*       -- Sets the noncurrent flag on the lesson's compilation entry so that */
/*          MEBuilder will know that the lesson must be recompiled before      */
/*          executing certain lesson-related commands.                         */
/*    mebuildLIB_mark_successful_compilation(+Lesson)                          */
/*       -- Informs the library that Lesson has successfully compiled and the  */
```

```
/*          current compilation resides in memory.                              */
/*    mebuildLIB_lesson_is_outdated(+Lesson)                                    */
/*       -- Performs a date check on all dependent entities to ensure that the  */
/*          lesson is based on the most up to date information.  If a task or   */
/*          class has been updated since the last save of the lesson, then the  */
/*          lesson is considered untrustworthy.                                 */
/*    mebuildLIB_compilation_is_current(+Lesson)                                */
/*       -- Performs a query on whether the compilation is current.             */
/*    mebuildLIB_set_library_lesson_dependency(+Lesson,class|task,+NewDependent) */
/*    mebuildLIB_remove_library_lesson_dependency(+Lesson,class|task,           */
/*                  +OldDependent)                                              */
/*       -- Sets and removes a class or task from the dependency list of the    */
/*          lesson.                                                             */
/*    mebuildLIB_get_compilation_file_name(+Lesson,-FileName)                   */
/*       -- Returns the standard file name for the compiled Lesson.             */
/*    mebuildLIB_get_compiled_lessons(-CompiledLessons)                         */
/*       -- Returns all loaded lessons whose compilations are current.          */
/*                                                                              */
/********************************************************************************/
```

TAB 7. METUTOR VERSION 29 SOURCE

```
/*************************************************************************/
/* Means-Ends Tutoring Platform -- Version 29    (METUTOR)               */
/* Original and versions 1 through 27 written by Professor Rowe          */
/* Version 28 and 29 (MEBuilder interface version) -- by Tom Galvin      */
/*************************************************************************/
/*                                                                       */
/*     METutor program provides problem-independent code for performing  */
/* means-ends tutoring: tutoring for learning of sequences modelable by  */
/* means-ends analysis.  This version for Quintus Prolog 3.0 and works   */
/* with graphics interfaces.                                             */
/*                                                                       */
/*     Version 29 takes the following facts produced by MEBuilder compiled */
/* lessons and runs the lesson.  NOTE:  *All* the below facts are compiled */
/* using dynamic modules -- so the module name must be used to access all */
/* the below facts except for those in the METutor environment.          */
/*                                                                       */
/*     The concept of the student's role still has to be programmed in, as */
/* the present example simply handles the case where the student subsumes */
/* the null agent argument.  Minor fix to bring it in line with the rest */
/* the MEBuilder system.                                                 */
/*                                                                       */
/*     METutor version 29 uses a workbook-like structure where the student */
/* enters the lesson and the lesson contains various problems or exercises */
/* which he needs to do.  Currently, version 29 only provides navigational */
/* frameworks in which a student can simply run any problem at will and no */
/* data about completion or non-completion of the problems are retained.  */
/* Among the items needed in future are:                                 */
/*     -- Student progress management.  Finish problem 1 before going on, */
/*        for example.                                                   */
/*     -- Course of instruction modeling.  Finish the lesson -- or master */
/*        the material -- and link into a new lesson.                    */
/*     -- Student modeling.  Beginners vs. Experts                       */
/*                                                                       */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* METutor's Basic Lesson Data Structure                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                       */
/*     METutor lessons contain several problems which are based on iden- */
/* tical sets of basic means-ends facts.  In order to save database space, */
/* the lesson is stored primarily in macro form wherever possible.  The  */
/* lesson is then macro-expanded into a problem module based on the spec- */
/* ific domains associated with the problem.  The lesson compiler module */
/* has a full description of the macro condensed version of the means-ends */
/* database.                                                             */
/*                                                                       */
/*     These are the facts that are not condensed -- therefore not macro */
/* expanded during problem initialization.                               */
/*                                                                       */
/*  lesson(<name>).                                                      */
/*     Simply a tag name for the lesson used during the welcome section. */
/*                                                                       */
/*  lesson_intro(<text>).                                                */
/*     Prolog fact buffer which contains lesson introductory text for the */
/*     student.                                                          */
/*                                                                       */
/*  problem(<number>,<name>,<student role>).                             */
/*     Provides a tag name for the problem.                              */
/*                                                                       */
```

```
/*  problem_intro(<number>,<text>).                                          */
/*     Prolog fact buffer which contains problem introductory text for the */
/*     student.                                                             */
/*                                                                          */
/*  problem_domain(<number>,<domain>,<range>).                              */
/*     Used to specify the particular elements of a domain for a partic-    */
/*     ular problem.                                                        */
/*                                                                          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* METutor's Macro Expanded Problem Data Structure                          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                          */
/*     The following are the facts that are macro-expanded into the prob-  */
/* lem module for use in the actual METutor session.  The <agent> argument */
/* is "student" for those facts that are related to the student's process. */
/*     The macro form and the expanded forms are given for each.            */
/*                                                                          */
/* PROBLEM-SPECIFIC METUTOR FACTS:                                          */
/*                                                                          */
/*  Macro:   start_state_t(<problem number>,<template>,<macro start>).      */
/*  Expand:  start_state(<start state>).                                    */
/*     Provides the specific problem's initial conditions.                  */
/*                                                                          */
/*  Macro:   goal_t(<problem number>,<agent>,<template>,<macro goal>).      */
/*  Expand:  goal(<agent>,<goal>).                                          */
/*  Macro:   goal_t(<problem number>,<template>,<macro goal>).              */
/*  Expand:  goal(<goal>).                                                  */
/*     Lists of properties which correspond to the starting conditions and  */
/*     objectives provided via MEBuild for when the student or agent's      */
/*     task is completed.                                                   */
/*                                                                          */
/*  Macro:   global_t(<problem number>,<template>,<macro goal>).            */
/*  Expand:  end_of_problem(<goal>).                                        */
/*     This defines when the simulation ends.  It is not required, and if   */
/*     it is omitted, then the problem ends when the student's goal has     */
/*     been reached.  If an explicit global_t fact exists, then even if     */
/*     the student's goal is completed, the simulation will continue.  The  */
/*     actual end is achieved when both the global_t condition and the      */
/*     student's goal conditions are achieved.                              */
/*                                                                          */
/* PROBLEM NON-SPECIFIC MEANS-ENDS FACTS:                                   */
/*                                                                          */
/*  Macro:   recommended_t(<role>,<template>,<macro difference>,            */
/*                         <macro operator>).                               */
/*  Expand:  recommended(<agent>,<difference>,<operator>).                  */
/*     Gives an operator recommeded to a achieve a particular set of        */
/*     facts different from the current state; conditionlist are facts      */
/*     that must be present in the current state.                           */
/*                                                                          */
/*  Macro:   precondition_t(<role>,<template>,<macro operator>,             */
/*                          <macro context>,<macro precondition>).          */
/*  Expand:  precondition(<agent>,<operator>,<context>,<precondition>).     */
/*     Gives facts required by operator in order to be used.  The <context  */
/*     list describes unique conditions under which various preconditions   */
/*     may hold.                                                            */
/*                                                                          */
/*  Macro:   addpostcondition_t(<role>,<template>,<macro operator>,         */
/*                              <macro context>,<macro add list>).          */
/*  Expand:  addpostcondition(<agent>,<operator>,<context>,<add list>).     */
/*     Gives facts added by the application of the operator.  <context>     */
/*     has same meaning as for precondition.                                */
```

```
/*                                                                    */
/*  Macro:   deletepostcondition_t(<role>,<template>,<macro operator>,  */
/*                                 <macro context>,<macro delete list>). */
/*  Expand:  deletepostcondition(<agent>,<operator>,<context>,          */
/*                               <delete list>).                       */
/*     Gives facts deleted and added by the application of the operation. */
/*     <context list has the same meaning as for precondition.         */
/*                                                                     */
/* TUTOR SUPPORTING FACTS.                                             */
/*                                                                     */
/*  Macro:   singular_t(<template>,<macro object>).                    */
/*  Macro:   plural_t(<template>,<macro object>).                      */
/*  Expand:  singular(<object>).                                       */
/*  Expand:  plural(<object>).                                         */
/*     Used to override the defaults concerning the conjugation of the */
/*     verb "to be" for those objects whose name doesn't follow standard */
/*     pluralization.                                                  */
/*                                                                     */
/*  Macro:   randchange_t(<template>,(<macro operator spec>),          */
/*             <macro context>, <macro delete>, <macro add>,           */
/*             <probability of occurrence>,<macro text>).              */
/*  Expand:  randchange((<operator spec>),<context>,<delete>,<add>,    */
/*             <probability of occurrence>,<text>).                    */
/*     Defines random events that are triggered by <operator> under the */
/*     conditions defined in <context>, or are a random initial condition */
/*     of the problem as defined by "init".  <probability of occurrence> */
/*     is a value from 0.0 to 1.0 which describes the chances of the   */
/*     event taking place given that the context is met.  <delete facts> */
/*     and <add facts> are similar to that of the postcondition process. */
/*     <message to student> is only printed in the event occurs.  The  */
/*     <context>, <delete facts>, and <add facts> may be empty.        */
/*     <macro operator spec> is one of <macro operator>, any_op, init, */
/*     or init(<problem number>).  <operator spec> is one of operator, */
/*     any_op, or init.  "init" and "init(problem number)" refer to ran- */
/*     dom start state information.  "any_op" refer to random changes  */
/*     that occur regardless of the operator used (the discriminating  */
/*     factor thus is the context).                                    */
/*                                                                     */
/* LESSON DISPLAY FACTS (TEXT INTERFACE):                              */
/* NOTE:  See graphicsflag below.  These facts are used when graphicsflag */
/*        is not set.                                                  */
/*                                                                     */
/*  Macro:   apply_text_t(<template>,<macro operator>,<macro context>, */
/*             <macro text>).                                          */
/*  Expand:  apply_text(<operator>,<context>,<text>).                  */
/*     The <text> is printed to the user whenever the given <operator> is */
/*     applied under the context of <context list>.  These are directly */
/*     aligned with the addpostcondition facts.                        */
/*                                                                     */
/* NOTES ON TEMPLATES:                                                 */
/*     The <template> can contain elements of any of the following forms: */
/*        <domain>     -- which expands to one element of a domain.    */
/*        some(<domain>) -- same as <domain>.                          */
/*        forall(<domain>) -- expands to all members of the domain.    */
/*     The "some" and "forall" quantifiers are somewhat misleading in that */
/*     they really mean "any" and "all".                               */
/*                                                                     */
/*                                                                     */
/*                                                                     */
/* LESSON DISPLAY FACTS (GRAPHICS INTERFACE):                          */
/*     Currently the graphics interface is not usable since megraph is set */
```

```
/*     for lessons compiled in module user -- not in a dynamically        */
/*     compiled module.                                                   */
/*                                                                        */
/*  graphicsflag.                                                         */
/*     Activates METutor's graphical interface.                           */
/*                                                                        */
/*  nocolorflag.                                                          */
/*     Operates black/white on a color terminal.                          */
/*                                                                        */
/*  bmap(<fact>, <conditions>, <picture-filename>, <x-coordinate>,        */
/*       <y-coordinate>, <width>, <height>, [<color>]).                   */
/*     If graphicsflag set, gives the name of a file holding a bitmap     */
/*     portraying the given fact when the conditions hold; coordinates are */
/*     upper left corner of place where bitmap is put; width and height   */
/*     are the size of the bitmap.  Bitmaps are optional for a fact, and  */
/*     there can be multiple bitmaps all displayed for the same fact.     */
/*                                                                        */
/*  text(<fact>, <conditions>, <text-string>, <x-coordinate>,             */
/*       <y-coordinate>).                                                 */
/*     If graphicsflag set, and context applies, writes that text at that */
/*     place on the screen                                                */
/*                                                                        */
/* METUTOR CACHED FACTS:                                                  */
/*     These facts are cached by lesson.                                  */
/*                                                                        */
/*  top_goal(<goal>).                                                     */
/*     Same as the goal fact.  Cached per lesson.                         */
/*                                                                        */
/*  top_solution(<list of operators>).                                    */
/*     This is a list of operators which METutor has determined is the    */
/*     most direct solution.  Cached per lesson.                          */
/*                                                                        */
/*  current_state(<state>).                                               */
/*     List of properties which indicate the present situation.           */
/*                                                                        */
/*  last_state(<state>).                                                  */
/*     The previous value of current_state/1.                             */
/*                                                                        */
/*  op_list(<global list of operators>).                                  */
/*     This is a list of all the operators available in the lesson.       */
/*                                                                        */
/*  solution(<agent>, <state>, <goal>, <oplist>, <goal state>).           */
/*     This is a cached solution to a subproblem of the lesson.  Used to  */
/*     save time in the calculation of future solutions.                  */
/*                                                                        */
/*  unsolvable(<agent>,<state>,<goal>).                                   */
/*     Indicates that <goal> cannot be reached from <state> by <agent>.   */
/*     Used to save time against computing known unsolvable problems.     */
/*                                                                        */
/*  session_num(<n>).                                                     */
/*     This is the nth run that the student has started.  (Counter which  */
/*     is maintained in the Prolog Utilities code.                        */
/*                                                                        */
/*  error_num(<n>).                                                       */
/*     The student has committed n-1 errors during the nth session.  (Also */
/*     a counter which is maintained in the Prolog Utilities code.        */
/*                                                                        */
/*  student_error(<session number>, <error number>, <student operator>,   */
/*                <tutor chosen operator>, <state>, <goal>).              */
/*     Stores the information about the nth error in the mth session.  The */
/*     <state> and <goal> are the current_state and the lesson objectives, */
```

```
/*     while <student operator> indicates the student's choice of operator */
/*     and <tutor chosen operator> was the preferred choice by METutor.    */
/*                                                                         */
/* METUTOR ENVIRONMENT FACTS:                                              */
/*                                                                         */
/*  debugflag.                                                             */
/*     If asserted, debugging info is printed during means-ends analysis   */
/*                                                                         */
/*  studentflag.                                                           */
/*     If asserted, does not print info on possible teacher errors.  This  */
/*     flag should be set when the student is running the program.  (The   */
/*     flag is set automatically when MEBuilder is running.                */
/*                                                                         */
/*  stop_time(<time>).                                                     */
/*     Busy wait indicator.  Tells the student "I am thinking..." during   */
/*     extended periods of calculation.                                    */
/*                                                                         */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Exported predicates                                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                         */
/*  run_lesson/run_lesson(<Lesson>).                                       */
/*     Runs the lesson loaded into the dynamic module <lesson module>.     */
/*     'user' is assumed if the argument is left off.  (HINT:  NEVER use   */
/*     'user'.  Always use a dynamic module in order to prevent problems   */
/*     regarding dynamic assertions and abolishments.)                     */
/*     For the graphics interface, you must have the lesson in the user    */
/*     module in Prolog and MUST use the non-argument run_lesson. (This    */
/*     will be corrected eventually.)                                      */
/*                                                                         */
/*************************************************************************/
```

APPENDIX B. MEBUILDER USER'S MANUAL

The manual, minus appendices, enclosed here is the same one provided to the students in the experiment discussed in Chapter VI. The following are the sections of the manual:

Tab 1. Introduction
Tab 2. MEBuilder's Interface and Environment
Tab 3. Library Facilities
Tab 4. Step One -- Designing an Object
Tab 5. Step Two -- Designing a Task
Tab 6. Step Three -- Designing a Lesson

TAB 1. INTRODUCTION

1. General.

a. *About the Manual.* This user's manual is the first draft for the MEBuilder lesson authoring system for METutor, specifically written for lab experimental purposes. It currently provides only a basic introduction to MEBuilder data structures and a reference guide for commands for the version dated 31 August 1994. Comments and suggestions are welcome.

b. *Files.* MEBuilder is a lesson authoring system written for METutor versions 29 and beyond. It is released in executable form and is available in the file **~galvint/mebuild/MEBuilder**. You are free to copy the file into your own directory to use (it is about 1.6MB large). *Please do not run MEBuilder in the galvint directory. You must run it in from your directory otherwise MEBuilder will not be able to write to the library file.* Similarly, METutor is available from **~galvint/mebuild/METutor**.

2. Special Features of MEBuilder.

a. *Library.* In the directory you are using, MEBuilder will set up a local library directory in **./lib**. In this directory will be all the data files of the lesson material you will produce. The directory will contain a special file, called **mebuild.lib** which contains data on all the files in the directory. Please do not use **./lib** for any purpose other than MEBuilder sessions.

b. *METutor Interface.* METutor is directly accessible from within MEBuilder using the ***run lesson*** command. This allows you to test a completed lesson without having to run a separate METutor session.

c. *Object-Oriented Structures.* MEBuilder uses an object-oriented system which allows you to re-use objects you create for use in multiple lessons. MEBuilder employs both generalization and aggregation principles along with part-kind inheritance. Single inheritance is the only type permitted, however.

3. Three-Layered Lesson Design.

When designing a lesson, you will do so in three steps. MEBuilder lesson material is constructed using a "bottom-up" approach -- meaning that you will start at the lowest level and end with the overall lesson. This bottom-up approach will be replaced in future versions with a more top-down approach.

a. *Design the objects.* Objects are the props and characters that the student will manipulate in the lesson. The student will describe the basic properties of the object and its behaviour.

b. *Design the tasks.* A task is a sequence of operations that take the student from some given condition to a specific goal.

c. *Design the overall lesson.* A lesson is a workbook of exercises. In each exercise, you will describe a scenario for the student and the goals that the student must achieve based on the tasks. Exercises may increase in scope or difficulty.

4. Layout of this Manual.

The manual is broken into five main sections -- one for the MEBuilder user interface, one for the library and one for each of the above lesson design steps. Each section contains a reference listing of the available commands along with some examples of the commands in use. For additional help, you may use the ***help*** command in MEBuilder. Appendix 1 contains a complete command reference. Appendix 2 contains a documented script run which takes you through all the fundamental steps for constructing a simple preflight tutor.

TAB 2. MEBUILDER'S INTERFACE AND ENVIRONMENT

1. Command Line Format.

MEBuilder is presently a pure character-based interface, which means that all inputs are from the keyboard and at present there is little support for graphics. The commands are one or two words long, and many have parameters which must be supplied.

In the appendix, you will note that the commands are listed in ***bold italics*** and arguments are listed in brackets. These are arguments which MEBuilder must have in order to process the command. however you have the choice of supplying them on the command line or specifying them when MEBuilder responds with a query. For example, the ***load object*** command takes a single parameter, that of **named** <*object*>. You may invoke the command in the following ways:

MEBUILD>**load object named my object**
-or-
MEBUILD>**load object**
Load which object?**my object**

Important Note: At any time while a command is invoked and you are being asked a question, you can return to the prompt by typing ***abort***.

2. MEBuilder's Command Loop Hierarchy.

MEBuilder's user interface is divided into four main loops. Upon executing MEBuilder, you enter the Main Loop. Certain commands access the other loops. In order to pop out of a given loop, you use the quit command. Using quit from the Main Loop exits MEBuilder.

a. *Main Loop*. The Main Loop represents the highest level interface command loop provided by MEBuilder. It provides access to MEBuilder's library functions and access to MEBuilder's object definition commands. The library functions include loading and saving commands for objects, tasks and lessons. The object definition commands include those that create and manipulate the object data structure.

The Main Loop prompt is MEBUILD>.

b. *Task Loop*. The task loop provides all of the commands to the user for defining and manipulating tasks. The task loop also imports various view commands from the main loop for classes and the library so the
user may query information about them. The task loop is invoked with one and only one task. This means that if you wish to work on a different task, you must exit the task loop using the ***quit*** command and reinvoke the loop with the desired task.

The task loop's prompt is [TASK:*task name*]> where the task being worked on is in place of *task name*. The task loop is accessed via the ***create task*** and ***work on task*** commands.

c. *Lesson Loop*. The lesson loop provides all of the commands to the user for defining and manipulating lessons. The lesson loop also imports various view commands from the main loop for classes and the library so the user may query information about them. The lesson loop also contains the commands to access METutor.

The lesson loop is invoked with one and only one lesson. This means that if you wish to work on a different lesson, you must exit the lesson loop using the ***quit*** command and reinvoke the loop with the desired lesson.

The lesson loop's prompt is [LESSON:*lesson name*]> where the lesson being worked on is in place of *lesson name*. This loop is accessed by the ***create lesson*** and ***work on lesson*** commands.

d. *Problem Loop*. The problem loop provides all of the commands to the user for defining and manipulating problems within a lesson. The problem loop also imports various view commands from the lesson loop for classes and the library so the user may query information about them.

The problem loop is invoked with one and only one problem. This means that if you wish to work on a different problem, you must exit the problem loop using the ***quit*** command and reinvoke the loop with the desired problem.

The problem loop's prompt is [PROB:*lesson name:problem number*]> where the lesson being worked on is in place of *lesson name* and the index number of the problem is in place of *problem number*. The loop is accessed by the ***work on problem*** command from within the Lesson Loop only.

3. Special Environment Features of MEBuilder.

a. *Autosave Capability*. MEBuilder comes with a built-in active autosave system which helps to preserve the session in case of ungraceful exit from MEBuilder. The autosave file (located in the working directory and called **autosave.meb**) is a pure database dump of all work being done with the Quintus Prolog dynamic modules preserved.

The autosave process occurs every tenth command in the Main Loop, Task Loop, Lesson Loop, or Problem Loop. If a complex session is taking place, this process can be quite slow -- hopefully future implementations will provide ways of speeding it up.

The autosave frequency and the destination autosave name can be set using the ***set autosave count*** and ***set autosave data*** commands. The ***view autosave data*** command will allow you to view the current autosave settings.

To restore an active session to the point of the last autosave, use the ***restore autosave file*** command.

b. *Help Facility*. At any time in the four main loops, you may invoked the help facility by selecting the ***help*** command. While in the help facility, you request information by providing the name of the command or topic you want at the MEBUILD HELP> prompt. You can get a list of help entries by typing ***help*** inside the Help facility. The ***quit*** command will return you to where you entered from.

TAB 3. LIBRARY FACILITIES

1. Introduction.

MEBuilder provides a basic library facility which helps track all the items created during MEBuilder sessions -- objects, tasks, lessons, and metutor-ready files. Items are tracked by date last saved and dependencies. Item dependencies indicate those items that must be in the database in order to use the item (example -- a task's dependencies are the objects involved in the task). The tracking of dependencies is used to auto-load everything needed in one stepand to ensure that changes in one item don't damage all other items that depend on it.

The library is stored in ./lib of the working directory andit contains a directory listing file (**mebuild.lib**) plus one filefor each item (object files have extension .cls, tasks .tsk,lessons .les, and METutor files .met). The listing file containsone entry per item indicating the name, file, etc. In future implementations, the listing file will also indicatelinks to other MEBuilder libraries. This will allow objects in other directories to be read-accessed by items in the local library.

2. Library Manipulation and Viewing Commands.

There are three things you can do with libraries -- create (or recreate) one, view the entries in one, or link in a new library. The options with these commands are currently very limited and will be upgraded in future.

a. *Clearing or resetting the library -- the* ***create library*** *command.* Currently, MEBuilder only recognizes './lib' as the library directory so running this command clears the library data file and starts the directory anew. Be careful when using this command -- be sure that an MEBuilder library does not exist (it will query to continue if one does).

b. *Viewing the Contents of the library -- the* ***view library*** *command.* This command will provide a by name listing of all the objects in the library -- which have been loaded and which have been modified. The list is alphabetized in order by objects, tasks, and lessons. (NOTE: Unfortunately the list is not run through **more** so it will likely scroll the screen. For non-xterm environments this might cause a problem. This feature is slated for future improvement).

c. *Linking together and accessing remote library information -- the* ***link library*** *command.* This command is not yet implemented. This command will allow you to access objects and tasks in multiple libraries. The key difference is that remote information is read-only, no library item can be mutually dependent, and a local definition of some library item takes precedence over items of the same name in the remote library.

d. *Cleaning out deleted information from the library -- the* ***purge library*** *command.* This command removes all entries from the library that have been marked for deletion by the ***remove object***, ***remove task***, and ***remove lesson*** commands. (This command is not yet implemented.)

3. Load and Save Commands.

Information about these commands are available in the reference section at the back of this manual and are discussed in detail in the other chapters of this text.

a. *Discussed in Section C (Objects) --* ***load object*** and ***save object***

b. *Discussed in Section D (Tasks)* -- ***load task***, and ***save task***

c. *Discussed in Section E (Lessons)* -- ***load lesson***, ***save lesson***, and ***save compiled lesson***

TAB 4. STEP ONE -- DESIGNING AN OBJECT

1. What is an Object?

An object is MEBuilder's representation for any entity that will exist in a lesson -- be it a prop or a character. The purpose of an object is to encompass the make-up and behaviour of these entities so they may be used in more than one lesson and the make-up and behaviour is guaranteed to be consistent.

Objects are abstract. When you develop a flashlight object, for example, you are defining behaviour true of all flashlights. This way, if you declare a lesson to have two flashlights -- a black and a silver -- the behaviour is consistent between the two.

Objects are represented as the collection of data relating to one ntity. These are:

-- Parent Object. Information about what type of object it is.
-- Components. Information about what other objects comprise this object.
-- Property Sets. Information about what states the object can be in and which states are mutually exclusive
-- Operations. Information about what operations can be performed on the object and what behavior is exhibited when done so
-- Summaries. Ways of summarizing a collection of states of an object in one term
-- Background Changes.
Information about behavior that an object may exhibit without an external stimulus

The three items above that are **required** for any object to be used in MEBuilder are the parent object, property sets, and operations. These three should be identified in order.

2. The Object Hierarchy -- Defining the Parent Object.

IMPORTANT NOTE: Before you declare a new object, ensure that its parent (defined below) is defined first and loaded into the current MEBuilder session.

Parent objects are used for specifying an object in which a new object shares data and behavior. The parent object is therefore a more general form of the new object. The opposite of parent object is child object.

For example, one may have a prop type named "car". One can then create "sedan" and "sports car" as new objects based on "car". "Car" becomes the parent object of "sedan" and "sports car" -- and the latter two inherit all the object definition data that exists in car. By "inheriting data", we mean that if a car has four wheels as components, then by the parent object relationship sedan automatically has the four wheels without you having to repeat that information when building a sedan.

Parent objects can be chained, meaning that if a "four-door sedan" was created, it inherits all information from car and sedan. Chained objects produce what we refer to as "ancestor/descendant" relationships. "Ancestor/descendant" can be used in lieu of "parent/child" as well.

Sedan can change some information in car also. For example, if cars have an engine, perhaps a sedan has a particular type of engine. So the engine component can be overwritten by sedan. Then "four-door sedan" inherits the sedan information first, not the car information.

In MEBuilder, every object is a descendant of either "prop" or "character". The parent object is specified upon creation of the object, and can be changed (This is strongly discouraged as it will disrupt the behaviour of any task or lesson that uses it). Objects can only have one parent.

The parent object is specified when the object is created using the ***create object*** command. MEBuilder will provide a list of all the objects currently loaded in the session plus the standard ones "prop" and "character". In order to change the parent object, you can use the ***modify parent object*** command.

3. Declaring Components.

After the object has been declared, the next thing you should do is declare any components it may have.

Component, in MEBuilder terms, is the way of describing both "a part of" and "has a" relationships between objects. For example, walking robots have x number of legs so each leg is "a part of" a walking robot. Or, a fire team member might have equipment that he would use in a firefighting problem. Therefore, the team member "has" equipment. With reference to characters, component relationships can probably be better described as "possessive" relationships and the component is the character's possession. The word possession is only used here for illustrative purposes -- component is used for both meanings.

MEBuilder allows you to identify component relationships between objects. Components are manipulated via the ***create component***, ***remove component***, and ***modify component*** commands. Apart from specifying the owning object and the component object type, you will need to specify:

-- A component name. For objects which have one of some type of component (cars have one engine, for ex.) you should use the object name as the component name. But, if the object has multiples of some component (cars have four wheels) then the component name must be unique ("left front wheel", "right front wheel", ..)

-- The tense of the component -- singular or plural. This ensures that the user output is correct in terms of matching verbs in natural language. MEBuilder will query the tense in the form of a question as to whether or not the name follows the "ends in s" rule. In other words, if it does not end in s it will assume it is singular and will query whether its assumption is correct.

The ***view component*** command will show the entire definition of the component, including its inheritance source (whether derived, inherited by parent object, or inherited by component).

4. Declaring Property Sets.

Property sets are used to describe states that an object may be in and the relationships (specifically mutual exclusion) among these states. A property set is defined as a set of properties that an object can only have one of at a given time. Examples of property sets:

-- A machine can be "on" or "off".
-- A streetlight can be "red", "green", or "yellow".

-- A student can be "present" or "not present".

Property sets always have a least two elements. A set that has one specified element "X" has an implicit element "not X". Property sets are described with the following information, and are manipulated via the ***create property set***, ***remove property set***, and ***modify property set*** commands.

-- The object being described.
-- A name which describes the set. The above three examples could be described in order as "switch position", "color", and "presence". IMPORTANT: The name must be one that can be used in natural language output because phrases such as "object's color is unknnown" may be shown to the student.
-- The members of the set (also called the domain of the set). Currently, only qualitative members are allowed. Quantitative members might be added at a later time.
-- Whether or not the state of an object is readily visible or is something that must be discovered. Examples include the "charge level" of a battery -- one doesn't know the charge level just by looking at it, one must perform an action to find out. This translates into information that a student is told he/she might not know when running the lesson.

Members of property set must be unique to the object. For example, the word "blue" cannot be used to describe both "color" and "mood". *You **cannot** begin the name of a property with the word **not**. **Not** is a reserved word. If you have a property set which has an x and not x relationship, you must specify only the x.*

The ***view object*** command will only show the name of the member property sets. To get the complete detailed definition of the set, use the ***view property set*** command. The output will also contain the inheritance source (whether derived, inherited by parent object, or inherited by component).

The following are suggestions for the naming of property set members. Although it seems puerile, the best way to name the property set members is to use the closest adjective form of the operation used. For example, a door object can be opened or closed. The operations would be "open door" and "close door". Similarly, a device that one can install or remove should be "installed" or "removed". Even though it sounds repetitve, it is easily for the student to grasp the direct relationship between his actions and the result.

5. Declaring Operations.

Ensure that you are declared all of your property sets first before entering this step.

Operations are the primitive methods by which the student or an agent can manipulate one or more objects. Tasks and lessons consist of sequences of these primitives. The intent of the operation is to encapsulate an event that takes precisely one turn to complete. In tasks and lessons, operations defined at the object level are called *primitive operations.*

Operations are described with several data items and are managed via the ***create operation***, ***remove operation***, and ***modify operation*** commands.

-- The direct object. The direct object is the thing that is the focus of the operation -- the primary item being manipulated. Can be either a whole object or a component of an object.
-- The indirect objects. These are objects which must be present in order to do this operation.
-- The operation name. Must contain direct object, but may contain any or all the indirect objects.
-- The intended effect. The primary or desired change of state in the direct object.

-- The preconditions. These are the conditions in which the direct object and indirect objects must be in for this action to be allowed.

-- The side effects. These are changes of state that any of the objects also realize different from the intended effect. These side effects *always* happen.

An example is "tighten the nut with the wrench" for the object nut. Nut is the direct object and wrench is an indirect object. However, let's say that bolt is also an indirect object that is unspecified in the operation name. The intended effect would likely be that the "nut is tightened". The preconditions would be that the "nut is on the bolt" and "wrench is serviceable". The side effects might be that the "bolt is not free".

It is important to note that the operation is defined only in instances where all the objects are used. Therefore, if a given task only uses nuts and bolts but not wrenches, the above operation cannot be used.

The following are the rules for operation names. The examples above show the pattern for operation names. The operations follow the convention of a verb phrase, followed by a direct object, trailed by a set of prepositional phrases containing the indirect objects. The verb phrase and the direct object are *required* elements of the operation name. However, there is one special case to be aware of. If the object is a character, and the direct object is the character object itself (not a possession of the character), then you should use the special form "***have <character> <operation name>***". What this will do is signal MEBuilder to strip off the "have" phrase when compiling any lesson with it. This way, the operation name correctly identifies the direct object and the end result operation is used by the student or agent in first person.

The ***view object*** command will only show a listing of operations by name for an object. To view the entire definition of the object, to include inheritance source, use the ***view operation*** command.

6. Declaring Summary Facts.

Summary facts are useful for complex objects in which you desire to help streamline the output. A summary fact is a single-phrase description of a collection of properties about one or more objects. It is primarily an interface tool which is used to "summarize" the state for the user.

Summary facts currently can only be defined for objects, however, they will soon also be definable for tasks and lessons. They consist of the following data and are accessible via the ***create summary fact***, ***remove summary fact***, and ***modify summary fact*** commands:

-- The object.

-- The name of the summary, which is constructed similar to a property set member: object is <summary descriptor>

-- The definition of the summary, which is a list of non-contradicting properties about the object.

An example of a summary fact for flashlight is "flashlight is working", defined as "flashlight"s *chassis is assembled*", "flashlight"s top is assembled", "flashlight"s batteries are working", "flashlight"s bulb is working". So with this summary fact, should all four defining members be true, then the four are not printed out to the student. Instead, just the summary "flashlight is working" is printed.

Summary facts are currently not available when using tasks -- either as defined among a set of objects or as a shortcut in building task definitions. This is an item for future implementation. The only time summary facts are used is in the METutor lesson itself.

The ***view object*** command will only list the object's summary facts by name. To get the complete definition of a summary fact, use the ***view summary fact*** command. The command will also show the inheritance source of the summary (whether defined or inherited from parent or component).

7. Declaring Background Changes.

Background changes model changes of state not caused by an external stimulus. That is to say that the object or objects change state on their own, due to the object being in some given condition (though not necessarily). An example of this is that whenever a streetlight is on, then if the streetlight is green then 10 turns
later it will turn yellow and 10 turns after that it will turn red.

There are several general models of background changes:

-- Progression. The object changes from state to state within a property set until the last is reached (which would normally imply some other event is to take place). An example of this is a ship with a hole in the hull that progresses through "no water", "some water", "lots of water", "full" at which point the ship sinks. Currently progression is forward only.

-- Loop. The object goes back to the beginning. The streetlight loops through "red", "green", "yellow". The loop can only go in one direction, it cannot be reversed at present.

-- Update. The object performs an operation. (Not yet implemented).

Background changes can be very complex, and there are many different options which are available for building them. They consist of the following pieces of data, and are manipulated using the ***create background change***, ***remove background change***, and ***modify background change*** commands.

-- A triggering condition list. The object would be subject to background changes while all of the conditions in the triggering condition list are true. If the list is empty, then the object will always be subject to the change.

-- The property set corresponding to the state changes.

-- Advancement method. How often or what probability will the next change occur.

The ***view object*** command will only display the name of the background changes in a listing. To get a complete definition of a background change, use the ***view background change*** command, which will also print the inheriteance source of the background change (derived or inherited by ancestor or component).

8. Library Management with Objects.

You may save your object at any time using the ***save object*** command. The ***load object*** command will load the object into the session, along with the entire parent and component class hierarchies. This last note is important since you may note that several objects are loaded in that you did not request loaded. This feature ensures that all inherited object definition data is available at all times while the object is being accessed.

You may also delete and restore objects from the library. The ***remove object*** command will mark an object for deletion, while the ***restore object*** unmarks it. Upon exit from MEBuilder, all objects removed from the library will be permanently purged. *Important: Once an object has been purged, all other objects, tasks, and lessons that use the object are invalid and must be reconstructed. Be sure you know what you are doing.*

(NOTE: These two commands are not yet implemented.)

Finally, to see if the object has a valid definition (that the modification or removal of data did not leave any undefined remnants, use the ***check object*** command.

TAB 5. STEP TWO -- DESIGNING A TASK

1. What is a Task?

A task is the fundamental building block of a lesson. It describes a single behaviour of a character (called an "actor") with respect to a defined starting point and a defined goal. This behaviour is described in terms of a "sequence" of primitive operations.

The task does several things. First, it establishes relationships among the primitive operations which the operations themselves do not cover. Second, it allows the teacher to identify and build alternate solutions to the problem to be eventually given to the student. Finally, it cross-checks the teacher's intent with the object
definitions to guaranteed correctness and consistency.

The task is made up of the following:

-- An actor. This is the one individual with primary responsibility for performing the task. The actor must be a character class.
-- A list of other objects required for the task. Some tasks may involve multiples of the same object. DO NOT treat components of an object as a separate object!
-- The initial conditions that each object is in at the beginning and the objectives that define when the task is complete.

Important point about tasks. Tasks are built in terms of a known start and a known finish. However, the task is designed so that the student can react to a state which is in the middle of the task or a state which is outside the bounds of the task. In each case, the task provides the underlying rules describing which operations can be applied and which cannot. Therefore, there is no restriction at the lesson level which prevents a lesson scenario from presented a situation that does not directly conform to the initial conditions of any task.

2. Properties of Tasks -- Steps and Step Dependency.

Tasks are described as a sequence of *steps* and each step consists of a *task operation.* The difference between a *task operation* and a *primitive operation* is that the *task operation* gains additional preconditions based on the sequence of steps in the task. So *task operations* are task-specific, whereas *primitive operations* are more general. Steps are identified by number, listed in front of the operation in brackets -- such as [3]. For commands that use <*step*> as an argument, it is the number in brackets that is required (this requires less typing than the entire operation name).

Tasks are not strictly linear, however. Some tasks can be performed in several different sequences of operations, usually because there are operations in the task that are completely unrelated. In this case, a single step may contain several "subprocedures":

```
[2]    open the widget
[3]    all of the following:
    [3a]    subprocedure:
        [3a1]yank the widget's red wire
        [3a2]yank the widget's yellow wire
    [3b]    subprocedure:
        [3b1]seal the room
```

```
          [3b2]deploy the bomb squad
[4]     yank the widget's blue wire
```

This implies that step [2] comes first, and that [3a1] must precede [3a2]. But it does not imply any special ordering between [3a1] or [3b1] other than both must be done before [4]. So the following are valid solutions -- 2-3a1-3b1-3a2-3b2-4, 2-3b1-3a1-3a2-3b2-4, etc.

There is an important principle that governs when steps can be moved around the procedure and where, and this is called *step dependency*. A task operation X is dependent on another task operation Y if and only if the intended effect or any side effect of the primitive operation Y is a precondition of the primitive operation X. What this means is that no matter how the teacher decides to describe the task, X *cannot precede* Y. Therefore, there will be limits to the options given to the teacher when moving task operations about the task. To see which operations are dependent on others, you may do the ***view step dependencies*** command.

During construction of the task, the ordering of steps adds more dependencies. Step [X+1] is always dependent on step [X], and similarly substep [XQg] is dependent on [XQf] for some subprocedure Q in step X. If step [X] is a divided step (meaning that is contains subprocedures), then step [X+1] is dependent on *all* of the last steps in each subprocedure of [X].

When you manipulate the task to change the ordering of steps etc., the options given are based on the primitive operation dependencies. Step dependency can be changed, the primitive operation dependencies cannot. In order to modify the latter, you must modify the object definitions appropriately.

3. Starting a New Task -- Naming Objects, and Setting the Intitial Conditions and Objectives.

To create a new task, use the ***create task*** command. This command will take you through a series of steps which will initialize the task into hopefully a solvable form.

a. *Naming the Objects Involved.* You will be asked for the actor in the task. The actor must be a character object (the generic character object is sufficient, and will probably be the one most commonly used). Then you will supply the object types for all the remaining objects. You are allowed to repeat objects. If you repeat objects, the second object of the same type will be object1. The third will be object2.

b. *Setting the Initial Conditions.* Initial conditions describe the initial states of the objects within a given task. These initial states are established in order to ensure that every property set associated with an object has a value assigned. After the task is defined, you may use the ***set initial conditions*** command to change them, and the ***view initial conditions*** command to view them. Notes about initial conditions:

(i) You can only choose one member of the property set to be in the initial conditions. You should choose the property set that satisfies the most general case of the task. It should not be necessary to create separate tasks based on an initial condition set with a variable member.

(ii) If a property set is declared hideable, then its value being known and/or unknown is considered a separate property set and must be set.

c. *Setting the Objectives.* Objectives define for some prop or character the point the goal of the task or lesson.After the task is initialized, the ***set objectives*** and ***view objectives*** commands are available. There are some subtle differences in the context of which prop or character:

(i) In the case of the prop, the objectives define what state the prop is in when the task or lesson is completed.

(ii) In the case of the character which is not the student's role, the objectives define the state that the character is always trying to achieve. It is best described as the state where he has no work to do.

(iii) In the case of the character which *is* the student"s role, the objectives define the state which signals the end of the lesson (or task).

Like the initial conditions, objectives are defined for each object in the task or lesson. The state for each
object should only list those properties absolutely necessary, you must explicitly identify those properties that are "don't-cares" -- MEBuilder uses the term "<property set> is immaterial." In order to prevent the unnecessary or unplanned exclusion of some solutions to the problem, make maximum (but well-planned) use of the "don't-care" case.

d. *MEBuilder Determines the Initial Solution.* MEBuilder will try to find *a* solution to the problem. Once it does, it will construct an initial procedure based on the premise that the solution it found it the only solution. The steps will be numbered [1] to [n]. The solution is based solely on the primitive operations, and it may not even be correct according to what you intended! You should accept the solution (forcing it to find a second solution could take quite a long time) and use the commands in the next section to manipulate it to the procedure you really want.

4. Modifying the Task -- Allowing for Multiple Solution.

There are several ways that you can manipulate the task once the first solution is established. It is important to note that all of these commands are restricted under the rules of dependency described earlier in this section.

a. *Declaring Subprocedures.* The ***find splits*** command will locate sets of operations that appear unrelated and could be made into subprocedures. The inverse of this is the combine step command. The ***combine step*** command does the inverse -- it takes two subprocedures (actually, the first step of each of two subprocedures) and combines them together into one sequence of actions.

b. *Swapping Steps.* Two adjacent steps may be in the incorrect order. As long as step dependency permits, you may use the ***swap step*** command to reverse them.

c. *Moving A Single Step Around the Task.* Sometimes the ***find splits*** command correctly identifies subprocedures except that one or two of the operations should be included or excluded. The ***move step*** command remedies these problems. In addition, this command allows naming the step as a single-action subprocedure, and you can create a set of permutable actions with this single-action subprocedure. For example if [4] and [5] are independent, then doing a ***move step number 5*** will allow you to merge [4] and [5] into [4a1] and [4b1]. This can be extended to add the one-action subprocedure to an already existing set of permutable actions.

d. *Declare Unordered Actions.* The ***move step*** is also used for this purpose. Steps may be declared unordered (and they will appear with an asterisk to the terminal). This same command can also be used to reorder the operation. Unordered actions are those which do not follow the strict ordering of the task except that they must precede the next step. For example, if step [6] is unordered it may be done at

any time (assuming its preconditions are met), but it must be done before step [7]. These greatly add to the flexibility of the lesson.

5. Viewing and Special Commands.

The ***reset task*** command is useful if you decide you wish to start over. MEBuilder will restore the initial single solution, clearing all subprocedure information.

The ***view task*** prints out the task in its present form. This is useful if you wish to get the current step arrangement.

The ***view situation*** command prints out the state that is true after a given operation has been performed. This is useful for uncovering reasons why a particular solution was declared valid when you feel it should not have been.

The ***modify step*** command allows you to add additional preconditions, side effects, and messages to the task. Once these are specified, the task is recalculated based on the changes. This is an especially important command if you wish to introduce probabilities into the problem (such as there is a 15% chance of a fire rekindling after you have extinguished it).

6. Library Management of Tasks.

The ***load task***, ***save task***, ***remove task***, ***restore task***, and ***check task*** commands are analogous to their object counterparts. The ***load task*** command will merely load the task into the session, it will not automatically invoke the Task Loop. check task will first perform a check object on all objects in the task.

In addition, in order to edit a task, you may use the ***work on task*** command to enter the Task Loop. This command will ensure the task is loaded first, and load it if necessary.

TAB 6. STEP THREE -- DESIGNING THE LESSON

1. What is a Lesson?

An MEBuilder lesson is effectively a workbook filled with individual problems, similar to an exercise section at the back of a chapter in a textbook. The problems should be designed to meet any or all of the following needs:

-- Cover different pieces of a topic, perhaps culminating with a problem that covers the whole thing.
-- Starting at an easy level and progressing to harder levels.
-- Demonstrating knowledge of lesson material in multiple scenarios (such as with different equipment or with different personnel available).

The workbook framework contains the main information necessary for MEBuilder to build an METutor lesson. These are:

-- A comprehensive cast of character roles and props with the associated tasks to be used.
-- An introduction text for the lesson itself.
-- A listing of problems.

2. What is a Problem?

A problem is a specific exercise. The student will do the problems when he runs the lesson. A problem contains a *scene* which is the scene presented to the student, and contains a *goal* which is what the student must achieve. A problem is constructed very similarly to a task in that a concrete cast and list of props is given. The problem also contains information about the role the student will play (if there is more than one character in the problem). Most lessons will have the student play the same role in all problems, however some might desire the student to play different roles.

3. Building a New Lesson.

Important: Before constructing a new lesson, you must have loaded into MEBuilder all of the objects and tasks you intend to include.

In order to make a new lesson, you may use the ***create lesson*** command from Main Loop prompt to make a new lesson. The command will take you through the process of identifying proper names for the cast and props along with identifying their types. This is the cast and props that will be present in all exercises.

MEBuilder will then identify the loaded tasks that correspond to the given set of objects. You may choose all of them or omit those not needed for the lesson. However, every object you specified must be includable in a task. In other words, if you specified an extinguisher object but omit the only loaded task that uses an extinguisher, then you must reconsider the assignment of objects and tasks. Ordering of tasks is important! The order you specify determines the defaults for the scene and the goal in cases when an object is to be employed in one or more task. The default scene for any object is the *initial conditions* of the first task that uses it. The default goal for any object is the objectives of the last task that uses it. These defaults will be relayed to you each time you declare a new problem.

Once the lineup is satisfactory, you will be in the Lesson Loop, the first thing you should do in create an introduction text for the lesson. This should be a general text which describes the overall goals of the lesson. Each problem will have its own introductory text. The ***edit lesson intro*** and ***view lesson intro*** commands are available. *NOTE: The **edit lesson intro** command uses emacs -- and at present this cannot be set.*

4. Defining a New Problem.

Problems are defined by the ***create problem*** command. This command will initiate a sequence of steps where you will specify the student's role and any cast members or props who are not to be included in this particular problem. Each problem is assigned a *problem number*, which is one more than the number of problems in the lesson. You also provide MEBuilder with a unique *problem name*, which is primarily used to help you identify the problem. In most commands in the Lesson Loop, you may identify a problem either by its number or its name.

The scene and the goal will default to the initial conditions and objectives of the tasks you specified according to the rules described above. MEBuilder will print out a list of those items as they are entered. Once the process is complete, you will be in the Problem Loop. Entering ***quit*** will pop you into the Lesson Loop.

After a problem has been created, you may modify the problem definition with the ***set scene*** and ***set goal*** commands. The ***view scene*** and ***view goal*** commands are analogous. The difference with the scene vice the initial conditions of a task, however, is that you may identify probabilities. This means that some state member in the scene is uncertain. You will be asked to identify those probabilities for each state member -- and those will become a random event in the compiled lesson.

You may edit and view the introductory text for the problem using the ***edit problem intro*** and ***view problem intro*** commands. These are exactly analogous to their lesson intro counterparts. As with the ***edit lesson intro*** command, emacs is the only editor accessible through MEBuilder.

In order to return to a problem from the Lesson Loop, you may ***work on problem***, specifying the problem's name or number.

5. Setting Problem Options (NOT YET IMPLEMENTED).

Options is a method of taking some of the parameters of a problem and adjusting them (overriding the original definition from the task) in order to change the difficulty of the problem.

None of the following have been fully implemented yet, however, these are the options planned:

-- Side effect blocking. All probabilistic side effects as defined in the task definitions are nullified. The intent is to provide a very simple problem with no surprises for the student (good for earlier problems).

-- Probability overrides. Overrides the percentage of some given side effect in a task definition. Intent is to perhaps increase the likelihood of something going wrong.

-- Background effect overrides. Overrides the percentages or count values in the background changes listed for some of the objects.

-- Character speed. Characters other than the student can takes action quicker. For example, a setting of 2 for the adversary of the student means that the adversary gets two moves per student's one move.

As they are implemented, entries for the associated commands will be added to the help system. Currently, the planned command is simply ***set options***.

6. Reordering and Deleting the Problems in the Lesson.

You are allowed to renumber the problems using the ***order problems*** command. This is provided since ***create problem*** only adds problems to the end.

In addition, problems may be deleted using the ***remove problem*** command.

7. Accessing METutor -- Compiling and Running Lessons from MEBuilder.

MEBuilder is a lesson authoring system that was built on top of METutor version 29, an Intelligent Tutoring System (ITS) shell. MEBuilder makes use of several METutor facilities while helping you build lessons.

Creating an METutor lesson from MEBuilder data is done through a "compilation" process, accessible via the ***compile lesson*** command. This command will performs checks on all the objects, tasks, and problems in the lesson and then translate the entire database into METutor form. The compiled lesson can be saved as an METutor runnable file using the ***save compiled lesson*** command.

You can invoke an METutor session on a lesson you have compiled by using the ***run lesson*** command in the Lesson Loop. You do not have to explicitly call ***compile lesson***, the ***run lesson*** command will do it automatically. Quitting from METutor will return you to the Lesson Loop.

8. Library Management of Lessons.

The ***load lesson***, ***save lesson***, ***remove lesson***, ***restore lesson***, and ***check lesson*** commands are available and mirror their object and task counterparts. In addition, the ***work on lesson*** command enters the Lesson Loop on a particular lesson. As with the task version, ***work on lesson*** will load the lesson first if it hasn't already been loaded into the session.

APPENDIX C. SAMPLE SCRIPT RUN WITH MEBUILDER

The following script run with inserted comments is the same sample script run given to the students in the experiment described in Chapter VI. The problem requires a pilot to execute the steps necessary to prepare a plane for takeoff. It is of similar complexity and size as those lessons used in the experiment. The script is divided as follows:

Tab 1. The Requirements and Design Phase
Tab 2. Building the Objects
Tab 3. Building the Task
Tab 4. Building the Lesson

TAB 1. THE REQUIREMENTS AND DESIGN PHASE

This appendix will present a sample session using MEBuilder to build a lesson from the ground up. That is, a requirement will be presented and discussed. We will then go through the process of identifying the objects and members of the objects, constructing associated tasks, and finally building a lesson workbook. We will then demonstrate the lesson compilation process and the METutor interface.

1. The Lesson, Pilot Training

The student is a pilot who is learning how to fly. His first lesson is in the process of prepping the aircraft for takeoff and for the basic communications with the air traffic control tower. At the start of the problem, all of the devices in the aircraft are turned off and the aircraft has not been preflight inspected.

This is a very simplistic description of the procedure and the objects involved in order to keep the scope of the problem reasonable for this appendix. There is a huffer present at the terminal to help start the engine. The aircraft has several subdevices associated with it. The aircraft has brakes, throttles, an engine, trim, nose wheel steering ("NWS"), external power hookups, and an auxiliary power unit ("APU") with its associated bleed air and generator. The brakes, trim, and NWS are unchecked. The aircraft has not been preflight inspected. External power is available but is off. The engine and APU (along with its associated components) are all off. The pilot has not had his flight plan cleared with tower, nor is he cleared for taxi nor takeoff. There is a wind sock at the end of the runway to inform the pilot of the wind speed and direction at takeoff. At the end of the task, the student should be airborne.

The sequence of steps is as follows:

(a) The pilot must do the following in order: inspect the aircraft, engage the external power, engage the huffer, start the engines, start the APU, and then engage the APU's generator and bleed air. The plane is own operating on its own.

(b) The pilot requests flight clearance while he is checking the NWS and brakes.

(c) He then disengages the huffer and the external power, requests taxi clearance and taxis to the runway.

(d) At the edge of the runway, he requests takeoff clearance while he is checking the wind sock and adjusting the trim on his aircraft.

(e) Once his takeoff clearance is granted, he pushes the throttles to max and flies the airplane!

2. Design the Objects and the Map the Scenario

For the purposes of this sample, we will only concern ourselves with the fundamental building blocks of objects -- components, property sets, and operations. We will not include summary facts and background changes nor will we employ an object hierarchy other than identifying props and characters. There are two steps to this process -- first identifying the makeup of the objects, then defining the behavior.

a. *Map the Scenario*

Generally speaking, most applications will not come in a nice tidy procedural listing such as that given above. The best thing to do for any application is start by identifying the specific

sequences of events that must occur and list then in order step (a) through (z). Grouping set sequences together such as (a) above is ideal. Devote a step to sequences of events where order is unimportant, such as (d) -- but be sure to identify what ordering is allowed and what is not. For example, in (d) checking the wind sock must come before adjusting the aircraft's trim. So the following are legal: clearance-wind sock-trim, wind sock-clearance-trim, and wind sock-trim-clearance.

b. *Identify and Design the Objects and Component Relationships*

There are several objects listed in the above scenario. However, as we identify these objects we must also identify those objects which are components of other objects or are possessions of characters. The following is the breakdown of objects in this task:

(1) the aircraft -- a prop, with the following components:
    (a) auxiliary power unit -- with the following components:
        (i) generator
        (ii) bleed air
    (b) brakes
    (c) engine
    (d) external power hookup
    (e) nose wheel steering
    (f) throttles
    (g) trim
(2) the pilot -- a character
(3) the huffer -- a prop
(4) the wind sock -- a prop

(We are not going to model the air traffic control tower as a separate object for this example.)

c. *Identify the Behaviour -- the Properties and the Operations*

The property sets and the operations go hand-in-hand. For each property described in the task, we might want to consider an operation that achieves it. The key for successfully doing this is to keep it simple. Do not include attributes that are redundant or unnecessary.

We will start with the pilot. The pilot is the one doing all the work, however most of the work is in changing the state of the aircraft, not the pilot. Three are three things the pilot does that clearly changes his own state -- requesting the three clearances. We say this because clearance to taxi, etc., is requested by and granted to the pilot, not the aircraft. Because the pilot is a character, the operations will begin with the key phrase "have pilot", which will be chopped off during the lesson so the student will view things first hand.

There are several ways to model the pilot's state. We could either model the three clearances as three different property sets or model all three as a single property set with a fourth member -- pilot is uncleared. Generally speaking the former method is preferable since the fact that a pilot is cleared for taxi and departure could be used in several contexts. Only in cases where all values a strictly mutually exclusive should they be combined.

When designing the objects, the best way to do it is to set up a table mapping properties to operations and operations to other data. Group possible property sets together and label them. The following is a good example:

Property	Operation to Achieve it	Precondition
flight clearance:		
pilot is cleared to depart	have pilot request flight clearance	
pilot is not cleared to depart	<none>	
taxi clearance:		
pilot is cleared to taxi	have pilot request taxi clearance	
pilot is not cleared to taxi	<none>	
takeoff clearance:		
pilot is cleared to take off	have pilot request takeoff clearance	
pilot is not cleared to take off	<none>	

Now, let's examine the task description and see what other information we can gather. For designing the operations, we must identify the preconditions. Using the scenario mapping, this is easy. For example, in order to request the flight clearance, the APU bleed air on the aircraft must be on. MEBuilder automatically assumes that the pilot is not cleared to depart so it is not necessary to include that precondition.

Property	Operation to Achieve it	Precondition
flight clearance:		
pilot is cleared to depart	have pilot request flight clearance	aircraft's bleed air must be on
pilot is not cleared to depart	<none>	
taxi clearance:		
pilot is cleared to taxi	have pilot request taxi clearance	aircraft's external power is off
pilot is not cleared to taxi	<none>	
takeoff clearance:		
pilot is cleared to take off	have pilot request takeoff clearance	aircraft must be on the runway
pilot is not cleared to take off	<none>	

In similar fashion, here are some the aircraft's properties and operations (We will not include all of them in the interest of space).

Property	Operation to Achieve it	Other Information
location:		
aircraft is at the terminal	<none>	
aircraft is on the runway	taxi the aircraft	pilot must be cleared to taxi
aircraft is airborne	fly the aircraft	aircraft's throttles must be maxed
NWS check status:		
aircraft's NWS is checked	check the aircraft's NWS	aircraft's bleed air must be on
aircraft's NWS is not checked	<none>	

Once you have built these tables for all the objects, you are ready to start the MEBuilder session.

TAB 2. BUILD THE OBJECTS

The best way to approach constructing the objects is to do the following in order:

-- Create all the objects
-- Connect all the components
-- Define all the property sets
-- Define all the operations

We will now begin an MEBuilder session. Notice that MEBuilder automatically creates a library directory for us (assuming this is our first session):

```
Script started on Tue Jul 26 10:33:14 1994
.alias: No such file or directory.
> ~galvint/mebuild/MEBuilder
+-----------------------------------------------------------+
| Means-Ends Virtual World and Lesson Builder --  MEBuilder |
| CPT Thomas P. Galvin, US Army,  Naval Postgraduate School |
+-----------------------------------------------------------+
|               Type "help" for assistance                  |
+-----------------------------------------------------------+
MEBuilder local library not found....creating...
MEBuilder local library loaded.
MEBUILD>
```

a. Creating the Objects. Now we will use the ***create object*** command to create all the objects (this includes all of the objects that will become components of aircraft). The following is a portion of the script.

```
MEBUILD> create object named pilot
The following are the available parent classes.
[1]   prop
[2]   character
Choose one of the above> 2
Class "pilot" is now defined.
MEBUILD> create object named huffer
The following are the available parent classes.
[1]   prop
[2]   character
[3]   pilot
Choose one of the above> 1
Class "huffer" is now defined.
MEBUILD> create object named wind sock
The following are the available parent classes.
[1]   prop
[2]   character
[3]   pilot
[4]   huffer
Choose one of the above> 1
Class "wind sock" is now defined.
MEBUILD> create object named aircraft
The following are the available parent classes.
[1]   prop
[2]   character
[3]   pilot
[4]   huffer
[5]   wind sock
```

```
Choose one of the above> 1
Class "aircraft" is now defined.
MEBUILD> create object named aircraft APU
The following are the available parent classes.
[1]   prop
[2]   character
[3]   pilot
[4]   huffer
[5]   wind sock
[6]   aircraft
Choose one of the above> 1
Class "aircraft APU" is now defined.
MEBUILD>
```

Note that all of the created objects become options for parent objects. In the same manner, the aircraft APU generator, aircraft APU bleed air, aircraft brakes, aircraft engine, aircraft external power, aircraft NWS, aircraft throttles, and aircraft trim objects are all created. Other commands useful here include ***view object*** and ***change parent object***.

b. Defining Components. We will demonstrate the process of combining a component object together. This is with the ***create component*** command. We will only show the construction of the aircraft APU with its generator and bleed air. As specified in the command reference, you do not have to provide the "named" and the "for" arguments on the command line -- MEBuilder will prompt those automatically. However, MEBuilder will always provide a menu for the component's type.

```
MEBUILD> create component named generator for aircraft APU
Choose the class of the new component from one of:
[1]   pilot
[2]   huffer
[3]   wind sock
[4]   aircraft
[5]   aircraft APU generator
[6]   aircraft APU bleed air
[7]   aircraft brakes
[8]   aircraft engine
[9]   aircraft external power
[10]  aircraft NWS
[11]  aircraft throttles
[12]  aircraft trim
Choose one of the above> 5
I assume "generator" is a singular name.Correct? [Yes/No] yes
Component "generator" defined for class "aircraft APU".
MEBUILD> create component named bleed air for aircraft APU
Choose the class of the new component from one of:
[1]   pilot
[2]   huffer
[3]   wind sock
[4]   aircraft
[5]   aircraft APU generator
[6]   aircraft APU bleed air
[7]   aircraft brakes
[8]   aircraft engine
[9]   aircraft external power
[10]  aircraft NWS
[11]  aircraft throttles
[12]  aircraft trim
Choose one of the above> 6
I assume "bleed air" is a singular name.Correct? [Yes/No] yes
Component "bleed air" defined for class "aircraft APU".
```

```
MEBUILD>
```

In similar fashion, the seven components of aircraft are attached. Once this is done, it would be wise to save all the objects. NOTE: saving the aircraft automatically causes all component objects to be saved also. You can check this using the ***view library*** command. As a check, here is what the ***view object*** command will produce on aircraft when this step is completed.

```
MEBUILD> view object named aircraft
Class "aircraft" is clean
     Dependencies: aircraft trim,  aircraft throttles,  aircraft NWS,
                   aircraft external power,  aircraft engine, aircraft
                   brakes,  aircraft APU,   and prop
Class "aircraft" is a prop class with superclass(es) "prop".
Class "aircraft" has no derived classes.
Components of class "aircraft":
[1]   "APU" of class "aircraft APU"
[2]   "brakes" of class "aircraft brakes"
[3]   "engine" of class "aircraft engine"
[4]   "external power" of class "aircraft external power"
[5]   "NWS" of class "aircraft NWS"
[6]   "throttles" of class "aircraft throttles"
[7]   "trim" of class "aircraft trim"
[8]   "APU's generator" of class "aircraft APU generator"
[9]   "APU's bleed air" of class "aircraft APU bleed air"
Property Sets of Class "aircraft":
<none>
Summary Facts of class "aircraft":
<none>
Property Display Data of Class "aircraft":
<none>
Operations of Class "aircraft":
<none>
Operation Display Text of Class "aircraft":
<none>
Background Changes of Class "aircraft":
<none>
MEBUILD>
```

Other useful commands relating to components are ***remove component***, ***view component***, and ***modify component***. Refer to the command reference.

c. Defining the Property Sets. The next step is to put in the first column of our property-operation table. The reason why we must enter all the properties first is because we cannot define an operation that involves a pilot and an aircraft unless the properties of both are defined. Therefore, it is good practice to ensure complete of the property sets before working on operations.

We will show the process for a one of the pilot's sets, then show a couple for the aircraft. The ***create property set*** command is used here. First the flight clearance for the pilot. NOTE: Notice that the "not" member is not given. Sets that have only one listed member automatically have a not second member. Do not give the "not" version!

```
MEBUILD> create property set named flight clearance for pilot
Type "exit" to leave.
New Property: pilot is -- cleared to depart
New Property: pilot is -- exit
Does this set correspond to information that could be hidden from the
student?  Answer yes or no.
```

```
>> no
Property set "flight clearance" defined for class "pilot".
MEBUILD>
```

Also note -- this example does not demonstrate hidden property sets. This is an advanced feature which will be demonstrated in a separate example in future.

Now, a demonstration of aircraft's location:

```
MEBUILD> create property set named location for aircraft
Type "exit" to leave.
New Property: aircraft is -- at the terminal
New Property: aircraft is -- on the runway
New Property: aircraft is -- airborne
New Property: aircraft is -- exit
Does this set correspond to information that could be hidden from the
student?  Answer yes or no.
>> no
Property set "location" defined for class "aircraft".
MEBUILD>
```

Now we will demonstrate the building of the NWS check status. Note first that the NWS check status is not a property set of the aircraft but of the component object! Properties describing components are always identified with their component object type. "Location" describes the whole aircraft, which is why it is a set of the aircraft.

```
MEBUILD> create property set for aircraft NWS
Please name the new property set> check status
Type "exit" to leave.
New Property: aircraft NWS is -- unchecked
New Property: aircraft NWS is -- checked
New Property: aircraft NWS is -- ecit
New Property: aircraft NWS is -- exit
Does this set correspond to information that could be hidden from the
student?  Answer yes or no.
>> no
Property set "check status" defined for class "aircraft NWS".
```

Oops! we mistyped "exit" and wound up with an extra property set member. To fix this, we will demonstrate the ***modify property set*** loop. Most of the modify commands in the object layer follow this type of a format. The definition of the object's property set is given and you select the specific attribute to change.

```
MEBUILD> modify property set named check status of aircraft NWS
Type "help" for help.  Type the label to change.
name   -- check status
domain -- unchecked, checked,  and ecit
hide   -- not_hideable
MODIFY PROPERTY SET> domain
Type "exit" to leave.
New Property: aircraft NWS is -- unchecked
New Property: aircraft NWS is -- checked
New Property: aircraft NWS is -- exit
Type "help" for help.  Type the label to change.
name   -- check status
domain -- unchecked and checked
hide   -- not_hideable
```

```
MODIFY PROPERTY SET> quit
Save changes to property set? yes
Property set modification completed.
MEBUILD>
```

In similar fashion we have defined the following property sets for all the objects:

Pilot:	Flight Clearance	"cleared to depart" and "not cleared to depart"
	Taxi Clearance	"cleared to taxi" and "not cleared to taxi"
	Takeoff Clearance	"cleared to takeoff" and "not cleared to takeoff"
Huffer:	Presence	"present" and "not present"
	Engagement	"engaged" and "disengaged"
Wind Sock:	Check Status	"checked" and "not checked"
Aircraft:	Location	"at the terminal", "on the runway", and "airborne"
Aircraft:	Preflight Completion	"preflight inspected" and "not preflight inspected"
Aircraft APU:	Switch Position	"off" and "on"
APU Bleed Air:	Switch Position	"off" and "on"
APU Generator:	Switch Position	"off" and "on"
Aircraft NWS:	Check Status	"unchecked" and "checked"
Aircraft Brakes:	Check Status	"unchecked" and "checked"
Aircraft Engine:	Running Status	"off" and "running"
A. External Pwr:	Usage	"available", "used", and "bypassed"
A. External Pwr:	Switch Position	"off" and "on"
Aircraft Thrott.:	Position	"off", "idle", and "max"
Aircraft Trim:	Check Status	"unchecked" and "checked"

Other useful commands for property sets are remove property set and view property set.

d. Defining the Operations. Now we will define the operations using the same table through the use of the ***create operation*** command. The process is very simple, but there are several steps involved. Here is the sequence of questions asked:

-- Name all the objects involved.
-- Provide the primary purpose, or "intended effect" of the operation. The intended effect is the property to the left of the operation name.
-- Provide the preconditions. These are given in the third column.
-- Provide the side effects. None of the operations have side effects. Side effects are other changes that occur that aren't the primary purpose of the operation.

For the first demonstration, we will do the one operation for the wind sock. The operation "check wind sock" will be used to achieve "wind sock is checked". Looking at the task, the precondition for this operation is that the "aircraft is on the runway".

One important note. Notice that the pilot object is identified as a necessary object even though there are no preconditions or side effects involving pilot. In general, if a specific character type (not the generic "character") is the one who will perform a given operation, then that character should always be included in the other objects list. This is to insure that the "check wind sock" operation involving a pilot is not confused with a "check wind sock" operation involving an air traffic control operator, for example.

First, identifying the objects and the operation name. Remember that the operation must follow this syntax -- <verb phrase> <object name or component name as direct object> <rest>. Since wind sock has no components, only "wind sock" may be used as the direct object.

```
MEBUILD> create operation for wind sock
You must now specify other objects needed to perform the operation.
You may repeat object types -- which implies a distinct object of same type.
IMPORTANT:  DO NOT INCLUDE COMPONENTS OF "wind sock"
UNLESS IT IS A COMPLETELY SEPARATE OBJECT.
[1]   pilot
[2]   aircraft
[3]   aircraft trim
[4]   aircraft throttles
[5]   aircraft NWS
[6]   aircraft external power
[7]   aircraft engine
[8]   aircraft brakes
[9]   aircraft APU
[10]  aircraft APU bleed air
[11]  aircraft APU generator
[12]  wind sock
[13]  huffer
Choose one or more of the above or "none"> 1 2
Name the operation: check wind sock
```

...next the intended effect.....

```
The following are the allowable intended effects:
[1]   wind sock is checked
[2]   wind sock is not checked
Choose one of the above> 1
```

...then the preconditions. You identify these one object at a time...

```
The following are the allowable preconditions for "pilot"
[1]   pilot is cleared to depart
[2]   pilot is not cleared to depart
[3]   pilot is cleared for taxi
[4]   pilot is not cleared for taxi
[5]   pilot is cleared for takeoff
[6]   pilot is not cleared for takeoff
Choose one or more of the above or "none"> none
The following are the allowable preconditions for "aircraft"
[1]   aircraft is preflight inspected
[2]   aircraft is not preflight inspected
[3]   aircraft is at the terminal
[4]   aircraft is on the runway
[5]   aircraft is airborne
[6]   aircraft's APU is off
[7]   aircraft's APU is on
[8]   aircraft's APU's generator is off
[9]   aircraft's APU's generator is on
[10]  aircraft's APU's bleed air is off
[11]  aircraft's APU's bleed air is on
[12]  aircraft's brakes is unchecked
[13]  aircraft's brakes is checked
[14]  aircraft's engine is off
[15]  aircraft's engine is running
[16]  aircraft's external power is available
```

```
[17]  aircraft's external power is used
[18]  aircraft's external power is bypassed
[19]  aircraft's external power is off
[20]  aircraft's external power is on
[21]  aircraft's NWS is unchecked
[22]  aircraft's NWS is checked
[23]  aircraft's throttles is off
[24]  aircraft's throttles is idle
[25]  aircraft's throttles is full
[26]  aircraft's trim is unchecked
[27]  aircraft's trim is checked
Choose one or more of the above or "none"> 4
```

...similarly the side effects. (There are no side effects possible for "wind sock" since there is only one property set for it, and a set cannot have members as both an intended effect and as a side effect. Therefore, "wind sock" is skipped.)

```
The following are the allowable side effects for "pilot"
[1]   pilot is cleared to depart
[2]   pilot is not cleared to depart
[3]   pilot is cleared for taxi
[4]   pilot is not cleared for taxi
[5]   pilot is cleared for takeoff
[6]   pilot is not cleared for takeoff
Choose one or more of the above or "none"> none
The following are the allowable side effects for "aircraft"
[1]   aircraft is preflight inspected
[2]   aircraft is not preflight inspected
[3]   aircraft is at the terminal
[4]   aircraft is airborne
[5]   aircraft's APU is off
[6]   aircraft's APU is on
[7]   aircraft's APU's generator is off
[8]   aircraft's APU's generator is on
[9]   aircraft's APU's bleed air is off
[10]  aircraft's APU's bleed air is on
[11]  aircraft's brakes is unchecked
[12]  aircraft's brakes is checked
[13]  aircraft's engine is off
[14]  aircraft's engine is running
[15]  aircraft's external power is available
[16]  aircraft's external power is used
[17]  aircraft's external power is bypassed
[18]  aircraft's external power is off
[19]  aircraft's external power is on
[20]  aircraft's NWS is unchecked
[21]  aircraft's NWS is checked
[22]  aircraft's throttles is off
[23]  aircraft's throttles is idle
[24]  aircraft's throttles is full
[25]  aircraft's trim is unchecked
[26]  aircraft's trim is checked
Choose one or more of the above or "none"> none
Operation has been added to class "wind sock".
MEBUILD>
```

...you will note that creating an operation is a very wordy process -- but because of the use of menus, there is little chance for error and little typing involved.

The second example is with the pilot. Because pilot is a character object, you will get a special message which describes the special syntax if the pilot is the direct object. We will only show the execution of the command up to the naming of the operation.

```
MEBUILD> create operation for pilot
You must now specify other objects needed to perform the operation.
You may repeat object types -- which implies a distinct object of same type.
IMPORTANT:  DO NOT INCLUDE COMPONENTS OF "pilot"
UNLESS IT IS A COMPLETELY SEPARATE OBJECT.
[1]   pilot
[2]   aircraft
[3]   aircraft trim
[4]   aircraft throttles
[5]   aircraft NWS
[6]   aircraft external power
[7]   aircraft engine
[8]   aircraft brakes
[9]   aircraft APU
[10]  aircraft APU bleed air
[11]  aircraft APU generator
[12]  huffer
[13]  wind sock
Choose one or more of the above or "none"> 2
This is a character object.  If itself is the direct object, you must
use the form "have <object> <operation>" where <operation> is not null.
Name the operation: have pilot request flight clearance
The following are the allowable intended effects:
```

...etc....

The final example involves the specification of a component direct object. This will be potentially very confusing because the general rule for where to place an operation is different than where to place a property set in a composite object. Operation go to the component only if the operation absolutely does not impact the whole object. For many of the aircraft operations, however, many of the operations changing one component have preconditions involving other components. Therefore, the below example for the "engage external power" operation is defined for the aircraft, not the aircraft external power. "external power", however, is a legal direct object because it is a component of aircraft.

```
MEBUILD> create operation for aircraft
You must now specify other objects needed to perform the operation.
You may repeat object types -- which implies a distinct object of same type.
IMPORTANT:  DO NOT INCLUDE COMPONENTS OF "aircraft"
UNLESS IT IS A COMPLETELY SEPARATE OBJECT.
[1]   aircraft trim
[2]   aircraft throttles
[3]   aircraft NWS
[4]   aircraft external power
[5]   aircraft engine
[6]   aircraft brakes
[7]   aircraft APU
[8]   aircraft APU bleed air
```

```
[9]   aircraft APU generator
[10]  pilot
[11]  wind sock
[12]  huffer
[13]  aircraft
Choose one or more of the above or "none"> 10
Name the operation: engage external power
The following are the allowable intended effects:
[1]   aircraft's external power is available
[2]   aircraft's external power is used
[3]   aircraft's external power is bypassed
[4]   aircraft's external power is off
[5]   aircraft's external power is on
Choose one of the above> 5
The following are the allowable preconditions for "aircraft"
[1]   aircraft is preflight inspected
[2]   aircraft is not preflight inspected
[3]   aircraft is at the terminal
[4]   aircraft is on the runway
[5]   aircraft is airborne
[6]   aircraft's APU is off
[7]   aircraft's APU is on
[8]   aircraft's APU's generator is off
[9]   aircraft's APU's generator is on
[10]  aircraft's APU's bleed air is off
[11]  aircraft's APU's bleed air is on
[12]  aircraft's brakes is unchecked
[13]  aircraft's brakes is checked
[14]  aircraft's engine is off
[15]  aircraft's engine is running
[16]  aircraft's external power is available
[17]  aircraft's external power is used
[18]  aircraft's external power is bypassed
[19]  aircraft's NWS is unchecked
[20]  aircraft's NWS is checked
[21]  aircraft's throttles is off
[22]  aircraft's throttles is idle
[23]  aircraft's throttles is full
[24]  aircraft's trim is unchecked
[25]  aircraft's trim is checked
Choose one or more of the above or "none"> 1
The following are the allowable preconditions for "pilot"
[1]   pilot is cleared to depart
[2]   pilot is not cleared to depart
[3]   pilot is cleared for taxi
[4]   pilot is not cleared for taxi
[5]   pilot is cleared for takeoff
[6]   pilot is not cleared for takeoff
Choose one or more of the above or "none"> none
The following are the allowable side effects for "aircraft"
[1]   aircraft is not preflight inspected
[2]   aircraft is at the terminal
[3]   aircraft is on the runway
[4]   aircraft is airborne
[5]   aircraft's APU is off
[6]   aircraft's APU is on
[7]   aircraft's APU's generator is off
[8]   aircraft's APU's generator is on
[9]   aircraft's APU's bleed air is off
[10]  aircraft's APU's bleed air is on
[11]  aircraft's brakes is unchecked
```

```
[10]  engage external power
[11]  start engine
[12]  disengage external power
[13]  engage APU's generator
[14]  engage APU's bleed air
Operation Display Text of Class "aircraft":
<none>
Background Changes of Class "aircraft":
<none>
MEBUILD> view object named wind sock
Class "wind sock" is clean
     Dependencies: prop
Class "wind sock" is a prop class with superclass(es) "prop".
Class "wind sock" has no derived classes.
Components of class "wind sock":
<none>
Property Sets of Class "wind sock":
[1]   check status
Summary Facts of class "wind sock":
<none>
Property Display Data of Class "wind sock":
<none>
Operations of Class "wind sock":
[1]   check wind sock
Operation Display Text of Class "wind sock":
<none>
Background Changes of Class "wind sock":
<none>
MEBUILD>
```

TAB 3. BUILD THE TASK

Building the objects is actually the most difficult part of the process. Once a good set of objects is built, building the task takes no more than a couple of steps. The process basically goes as follows:

-- Naming the task and telling MEBuilder all the objects in the task.
-- Telling MEBuilder what the state each object is in at the beginning of the task and what state constitutes the completion of the task.
-- MEBuilder generates one solution to the task and saves it as the only solution.
-- You tell MEBuilder what other solutions are allowed. At each step, MEBuilder will ensure that your other solutions in fact work.

The first three are all accomplished when you perform the ***create task*** command. Here is how this command works:

```
MEBUILD> create task named prep aircraft
For the primary actor, choose from one of the following:
[1]   character
[2]   pilot
Choose one of the above>
```

You will first note that the create task command will ask for an actor, which must be a character type. You can use the generic "character" or specify a specific character type that must be in the task. The only options are those loaded in the session -- so ensure you have loaded all your objects in the session first. We will select "pilot".

```
Choose one of the above> 2
List all of the other objects that are required for this task. If
there must be more than one, then repeat that item.  Ensure you
choose the most general object available.
[1]   pilot
[2]   huffer
[3]   aircraft
[4]   aircraft trim
[5]   aircraft throttles
[6]   aircraft NWS
[7]   aircraft external power
[8]   aircraft engine
[9]   aircraft brakes
[10]  aircraft APU
[11]  aircraft APU bleed air
[12]  aircraft APU generator
[13]  wind sock
Select one or more of the above or "none">> 2 3 13
Task "prep aircraft" is now defined.
Now working on task "prep aircraft".
Type "quit" to return to MEBUILD prompt.
```

For the follow-on part, we have identified that we need a huffer, wind sock, and an aircraft. Note that naming all the components of aircraft are unnecessary. If an aircraft and an aircraft engine are listed, then that constitutes a full aircraft and a separate engine.

Also, if needed, you could have specified a second pilot or multiples of any other object. If multiples of an object are specified, then they are referred to in the task as object, object1, object2, etc. (In the future, this will be changed to "second object", "third object", etc.).

Now, we will go through the process of identifying the initial conditions and the objectives of each object in the task. When you do a create task command, the default initial conditions are the first member of each property set. For sets of one member, constituting the X, not X case -- the not X is the default. For the objectives, the default is the last member of each property set and the X case holds for single member set.

For the pilot, the defaults are OK.....

```
The following are the current initial conditions for "pilot".
Indicate which ones you want changed:
[1]   pilot is not cleared to depart
[2]   pilot is not cleared for taxi
[3]   pilot is not cleared for takeoff
Choose one or more of the above or "none"> none
The following are the current objectives "pilot".
Indicate which properties you want changed.
[1]   pilot is cleared to depart
[2]   pilot is cleared for taxi
[3]   pilot is cleared for takeoff
Choose one or more of the above or "none"> none
```

...but for the huffer, we must make a couple changes. The default says that the huffer is not present, which is not true. What we do is identify the first property as needing to be changed. We select the correct property and then continue on.

```
The following are the current initial conditions for "huffer".
Indicate which ones you want changed:
[1]   huffer is not present
[2]   huffer is disengaged
Choose one or more of the above or "none"> 1
Choose the appropriate new initial condition:
[1]   huffer is not present
[2]   huffer is present
Choose one of the above> 2
Indicate which ones you want changed:
[1]   huffer is present
[2]   huffer is disengaged
Choose one or more of the above or "none"> none
```

Now, for the objectives we will change one of the properties also. The huffer, according to task, is disengaged when the task is finished. Well, let's say that we really don't care what condition the huffer is in so we will declare it as a don't care. The "is immaterial" option is the one we want.

```
The following are the current objectives "huffer".
Indicate which properties you want changed.
[1]   huffer is present
[2]   huffer is engaged
Choose one or more of the above or "none"> 2
Choose the appropriate new objective:
[1]   huffer is disengaged
[2]   huffer is engaged
[3]   huffer's engagement is immaterial
Choose one of the above> 3
```

```
Indicate which properties you want changed.
[1]   huffer is present
[2]   huffer's engagement is immaterial
Choose one or more of the above or "none"> none
```

Don't cares in the objectives are not printed to the student -- so the only thing that the student will see is that the huffer must be present when the task is complete. The following is the best rule for deciding don't cares -- an item should be a don't care if it meets the following criteria:

-- The property set value is not critical in defining the desired end result.
-- The property set value will only serve to confuse the student if it is listed (the objectives should be a minimal set).
-- The property set value is something that might change after the student has finished with it. (For example, a similar task might be set up such that an agent might move the huffer to another aircraft after the student is through with it).

The initial conditions of the aircraft are the default -- no changes needed (not preflight inspected, at the terminal, APU and its components are off, brakes unchecked, engine off, external power available and off, NWS checked, throttles off, and trim unchecked). We will now skip ahead to the point where the objectives of the aircraft are specified.

```
. . .
Indicate which properties you want changed.
[1]   aircraft is preflight inspected
[2]   aircraft is airborne
[3]   aircraft's APU's switch position is immaterial
[4]   aircraft's APU's generator's switch position is immaterial
[5]   aircraft's APU's bleed air's switch position is immaterial
[6]   aircraft's brakes are checked
[7]   aircraft's engine is running
[8]   aircraft's external power's usage is immaterial
[9]   aircraft's external power's switch position is immaterial
[10]  aircraft's NWS is checked
[11]  aircraft's throttles are full
[12]  aircraft's trim is checked
Choose one or more of the above or "none"> none
```

Numbers 3,4,5,8, and 9 are reasonable choices to omit from the objectives because the pilots main task is to check everything and get the plane in the air. The APU and external power don't tell the student anything directly perhaps. One could argue that #11 could be omitted as well. It all depends on what you want the student to see and which items are absolutely critical in defining the final objective (such as #2).

We will also skip the wind sock (IC=not checked, OBJ=checked). Once the four objects are completely done, we then enter the third step -- where MEBuilder attempts to find a solution. Here is the one it found:

```
Please wait....I am trying to solve the problem...
The following is my first attempt at solving the task.
[1]   conduct preflight inspection on aircraft
[2]   engage aircraft's external power
[3]   engage huffer
[4]   start aircraft's engine
```

```
[5]   start aircraft's APU
[6]   engage aircraft's APU's generator
[7]   engage aircraft's APU's bleed air
[8]   have pilot request flight clearance
[9]   disengage huffer
[10]  disengage aircraft's external power
[11]  check aircraft's NWS
[12]  check aircraft's brakes
[13]  have pilot request taxi clearance
[14]  taxi aircraft
[15]  have pilot request takeoff clearance
[16]  check wind sock
[17]  adjust aircraft's trim
[18]  shut off aircraft's APU
[19]  max aircraft's throttles
[20]  fly the aircraft
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
Action successfully added to the task.
```

What the "action successfully...." message means is that the task data structure is now set such that the above 20-step procedure is the only solution. We know, however, that this is not the case.

IMPORTANT: Troubleshooting the task if there is no solution found or the solution is wrong. Currently, MEBuilder has a very limited capability to identify specific reasons why a solution cannot be found. Remedies are underway. In the meantime, here are several steps to take if MEBuilder does identify such a problem.

1. Using the ***set initial conditions*** or ***set objectives*** commands, reduce the scope of the task to a specific segment (say, step (a) in the original task description). Locate the segment in which MEBuilder cannot find a solution and then check the operations in that segment.

2. Using the ***set objectives*** command, remove some of the don't care conditions and name a hard value as an objective. Sometimes a don't care will make the objective unachievable.

The next thing that happens is that the prompt has changed to the following:

```
[TASK:prep aircraft]>
```

This indicates that you are now in the Task Loop. There are different commands available here. In general, you can still use many of the view commands available from the Main Loop to query about objects and some of the attribute items. However, you cannot manipulate the object definitions -- while you are in the Task Loop all objects, other tasks, etc. are frozen.

Now, the first thing we will do is use the ***find splits*** command. This command looks through the task and decides if and which operators could be done in different orders. Currently, with the one solution, the task dictates that the request for flight clearance must precede the disengaging of the huffer -- which we know not to be the case. This command will identify this and propose a possible change:

```
[TASK:prep aircraft]> find splits
The following appear unrelated and could be parallel:
Sequence:
[1]   have pilot request flight clearance
Sequence:
[1]   disengage huffer
[2]   disengage aircraft's external power
[3]   check aircraft's NWS
[4]   check aircraft's brakes
Make them parallel sequences? yes
The following is the procedure as presently defined.
[1]    conduct preflight inspection on aircraft
[2]    engage aircraft's external power
[3]    engage huffer
[4]    start aircraft's engine
[5]    start aircraft's APU
[6]    engage aircraft's APU's generator
[7]    engage aircraft's APU's bleed air
[8]    all of these:
  [8a]   subprocedure:
    [8a1]   disengage huffer
    [8a2]   disengage aircraft's external power
    [8a3]   check aircraft's NWS
    [8a4]   check aircraft's brakes
  [8b]   subprocedure:
    [8b1]   have pilot request flight clearance
[9]    have pilot request taxi clearance
[10]   taxi aircraft
[11]   have pilot request takeoff clearance
[12]   check wind sock
[13]   adjust aircraft's trim
[14]   shut off aircraft's APU
[15]   max aircraft's throttles
[16]   fly the aircraft
```

Now look at the task construction. The steps have been renumbered such that step 8 now consists of a "split" -- two sequences of actions that must be completed before continuing on. So step 8a1 can be done anywhere among the steps in 8b so long as it follows 7 and precedes 8.

Another is found.

```
The following appear unrelated and could be parallel:
Sequence:
[1]   have pilot request takeoff clearance
Sequence:
[1]   check wind sock
[2]   adjust aircraft's trim
Make them parallel sequences? yes
```

```
The following is the procedure as presently defined.
[1]    conduct preflight inspection on aircraft
[2]    engage aircraft's external power
[3]    engage huffer
[4]    start aircraft's engine
[5]    start aircraft's APU
[6]    engage aircraft's APU's generator
[7]    engage aircraft's APU's bleed air
[8]    all of these:
  [8a]   subprocedure:
    [8a1]   disengage huffer
    [8a2]   disengage aircraft's external power
    [8a3]   check aircraft's NWS
    [8a4]   check aircraft's brakes
  [8b]   subprocedure:
    [8b1]   have pilot request flight clearance
[9]    have pilot request taxi clearance
[10]   taxi aircraft
[11]   all of these:
  [11a]  subprocedure:
    [11a1]  check wind sock
    [11a2]  adjust aircraft's trim
  [11b]  subprocedure:
    [11b1]  have pilot request takeoff clearance
[12]   shut off aircraft's APU
[13]   max aircraft's throttles
[14]   fly the aircraft
There are no possible parallel actions remaining.
[TASK:prep aircraft]>
```

At this point, if you study the task construction you will notice that it conforms to the original task specification -- meaning we are done! Two steps and that was it.

However, we all know that some task constructions are not quite this convenient. Therefore, we will demonstrate some of the other common options. The first item to note is that most of these commands use the step number as the argument. In order to keep up to date on step numbers -- use the ***view task*** command.

We will first demonstrate the ***move step*** command. This command is especially useful if you need to move a single step into or out of subprocedures, or declared it as an unordered action. Let's say that the "disengage huffer" operation really needed to precede the request for flight clearance -- we can move that step out of the subprocedure and establish it at step 8 -- the split becomes step 9:

```
[TASK:prep aircraft]> move step number 8a1
Here is what you may do:
[1]   Leave it alone
[2]   Move it out of subprocedure to in front of it.
[3]   Declare it doable *anytime* before "disengage aircraft's external
      power"
Choose one of the above> 2
Action successfully moved.
[TASK:prep aircraft]> view task
The following is the procedure as presently defined.
[1]    conduct preflight inspection on aircraft
[2]    engage aircraft's external power
[3]    engage huffer
[4]    start aircraft's engine
[5]    start aircraft's APU
```

```
[6]    engage aircraft's APU's generator
[7]    engage aircraft's APU's bleed air
[8]    disengage huffer
[9]    all of these:
  [9a]   subprocedure:
   [9a1]   disengage aircraft's external power
   [9a2]   check aircraft's NWS
   [9a3]   check aircraft's brakes
  [9b]   subprocedure:
   [9b1]   have pilot request flight clearance
[10]   have pilot request taxi clearance
[11]   taxi aircraft
[12]   all of these:
  [12a]  subprocedure:
   [12a1]  check wind sock
   [12a2]  adjust aircraft's trim
  [12b]  subprocedure:
   [12b1]  have pilot request takeoff clearance
[13]   shut off aircraft's APU
[14]   max aircraft's throttles
[15]   fly the aircraft
AUTOSAVING...please wait...Done
[TASK:prep aircraft]>
```

The ***move step*** command is invertible as well -- a second application of this command will allow you to put step 8 back where it was. Similarly, if you use the third given option to make the step unordered you can use move step to make it ordered again. The important thing to understand about this command is that the object definition restricts your options -- if you want to move the step somewhere that violates the object definition MEBuilder will not let you.

Let's say that we decide that we really didn't want the split in step 9 after all. The ***combine step*** command combines the subprocedures together. The arguments must be the first step in the two subprocedures to be combined.

```
[TASK:prep aircraft]> combine step number 9a1 with 9b1
Combination of subprocedures successful.
[TASK:prep aircraft]> view task
The following is the procedure as presently defined.
[1]    conduct preflight inspection on aircraft
[2]    engage aircraft's external power
[3]    engage huffer
[4]    start aircraft's engine
[5]    start aircraft's APU
[6]    engage aircraft's APU's generator
[7]    engage aircraft's APU's bleed air
[8]    disengage huffer
[9]    disengage aircraft's external power
[10]   check aircraft's NWS
[11]   check aircraft's brakes
[12]   have pilot request flight clearance
[13]   have pilot request taxi clearance
[14]   taxi aircraft
[15]   all of these:
  [15a]  subprocedure:
   [15a1]  check wind sock
   [15a2]  adjust aircraft's trim
  [15b]  subprocedure:
   [15b1]  have pilot request takeoff clearance
[16]   shut off aircraft's APU
```

```
[17]   max aircraft's throttles
[18]   fly the aircraft
```

You will notice now that the request for flight clearance is now step 12. Let's say that we wanted it in step 11 ahead of "check aircraft's brakes". The ***swap step*** command will allow you to swap adjacent steps X and Y if Y does not have to follow X due to X and Y's operation definition. For example, we cannot do 18 before 17, so doing ***swap step number 17 with 18*** will be disallowed. However, reviewing the operation definitions for 11 and 12 we find that we can swap them.

```
[TASK:prep aircraft]> swap step number 11 with 12
Combination of subprocedures successful.
[TASK:prep aircraft]> view task
The following is the procedure as presently defined.
[1]    conduct preflight inspection on aircraft
[2]    engage aircraft's external power
[3]    engage huffer
[4]    start aircraft's engine
[5]    start aircraft's APU
[6]    engage aircraft's APU's generator
[7]    engage aircraft's APU's bleed air
[8]    disengage huffer
[9]    disengage aircraft's external power
[10]   check aircraft's NWS
[11]   have pilot request flight clearance
[12]   check aircraft's brakes
[13]   have pilot request taxi clearance
[14]   taxi aircraft
[15]   all of these:
  [15a]  subprocedure:
    [15a1]  check wind sock
    [15a2]  adjust aircraft's trim
  [15b]  subprocedure:
    [15b1]  have pilot request takeoff clearance
[16]   shut off aircraft's APU
[17]   max aircraft's throttles
[18]   fly the aircraft
```

There are other useful commands such as ***modify step*** which allows you to add preconditions and side effects (especially probabilistic ones) and messages that are valid only within the context of this task. The command reference guide will be of more assistance.

TAB 4. BUILD THE LESSON

Now that the task is complete, we are ready to build the lesson. The lesson is set up as a workbook, a series of problems relating to the same theme. In this section we will demonstrate how one can set up three problems, where problems one and two relate to the two halves of the task and problem three is considered a comprehensive test.

The ***create lesson*** command behaves in a similar manner as ***create task*** in that a series of steps are accomplished:

-- Naming the lesson and identifying the cast and props
-- Naming the tasks to be used in the lesson

During the task construction phase, you were referring to the objects by type name. Now you are going to identify objects and give them proper names -- which will be used in referring to the lesson. For many simple applications, you will not need to identify any fancy names for the objects. A simple convention is to use "you" for the role of the student and "the <object>" for all the objects. But sometimes spicing up the affair can make the problem more fun -- like calling the aircraft "Bouncing Betty" a la World War II.

The important thing about the create lesson command is that you must specify the entire cast and props. As you create problems for the lesson, you will name objects and cast members not included in the problem (This is like naming the entire cast for a film and then leaving out some members in each scene).

Here is a script run of the create lesson command:

```
MEBUILD> create lesson named pilot training I
Choose all of the props and characters for the lesson.  If any
are omitted in any of the problems, you will specify those in
the problem.  Choices for commands are:
   add new X      -- add a member of object type #X
   remove item X -- remove #X from the current list
   clear          -- wipe out the list and start over
   quit           -- accept the list and continue
   abort          -- abort defining the lesson
These are currently in the lesson:
<none>
These are your choices for adding to the lesson:
[1]   character
[2]   wind sock
[3]   aircraft
[4]   aircraft trim
[5]   aircraft throttles
[6]   aircraft NWS
[7]   aircraft external power
[8]   aircraft engine
[9]   aircraft brakes
[10]  aircraft APU
[11]  aircraft APU bleed air
[12]  aircraft APU generator
[13]  buffer
[14]  pilot
CAST/PROPS>> add new 14
Give a name for the new pilot> captain jack
I assume "captain jack" is a singular name.  Correct? [Yes/No] yes
These are currently in the lesson:
```

```
[1]   Character: "captain jack" of "pilot"
These are your choices for adding to the lesson:
[1]   character
[2]   wind sock
[3]   aircraft
[4]   aircraft trim
[5]   aircraft throttles
[6]   aircraft NWS
[7]   aircraft external power
[8]   aircraft engine
[9]   aircraft brakes
[10]  aircraft APU
[11]  aircraft APU bleed air
[12]  aircraft APU generator
[13]  huffer
[14]  pilot
CAST/PROPS>> add new 3
Give a name for the new aircraft> the aircraft
I assume "the aircraft" is a singular name.  Correct? [Yes/No] yes
These are currently in the lesson:
[1]   Character: "captain jack" of "pilot"
[2]   Prop: "the aircraft" of "aircraft"
These are your choices for adding to the lesson:
[1]   character
[2]   wind sock
[3]   aircraft
[4]   aircraft trim
[5]   aircraft throttles
[6]   aircraft NWS
[7]   aircraft external power
[8]   aircraft engine
[9]   aircraft brakes
[10]  aircraft APU
[11]  aircraft APU bleed air
[12]  aircraft APU generator
[13]  huffer
[14]  pilot
CAST/PROPS>> add new 13
Give a name for the new huffer> the huffer
```

..... etc. as we add the other items (a huffer and a wind sock), we get:

```
These are currently in the lesson:
[1]   Character: "captain jack" of "pilot"
[2]   Prop: "the aircraft" of "aircraft"
[3]   Prop: "the huffer" of "huffer"
[4]   Prop: "the wind sock" of "wind sock"
These are your choices for adding to the lesson:
[1]   character
[2]   wind sock
[3]   aircraft
[4]   aircraft trim
[5]   aircraft throttles
[6]   aircraft NWS
[7]   aircraft external power
[8]   aircraft engine
[9]   aircraft brakes
[10]  aircraft APU
[11]  aircraft APU bleed air
[12]  aircraft APU generator
```

```
[13]  huffer
[14]  pilot
CAST/PROPS>> quit
```

Now, upon reaching this point you will be asked to supply the tasks to be used in the lesson. Only those tasks which have been loaded and whose required objects are a subset of the cast and props listed above will be included. In this exercise, only the "prep aircraft" task applies, so we show it.

```
When listing the tasks, specify the tasks according in the most
likely order that the student would apply them.  This will
establish the default scene and objectives for the problems.
The following loaded tasks are available:
[1]   prep aircraft
Choose one or more of the above> 1
Lesson "pilot training I" is now defined.
Work on lesson now? yes
Now working on lesson "pilot training I".
Type "quit" to return to MEBUILD prompt.
[LESSON:pilot training I]>
```

Important: Every object must have a task associated with it! No object may be idle!

Now, note that the prompt has changed again. Just as before with the tasks, the create lesson command brings you into the Lesson Loop. Once you are here, the things to do are the following:

-- Set up the introductory text using the edit lesson intro command.
-- Set up the problems.
-- Compile and run the lesson to test it out

We will skip the ***edit lesson intro*** command and show you the intro we put in using ***view lesson intro***.

```
[LESSON:pilot training I]> view lesson intro
The following is the introduction text for this lesson:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PILOT TRAINING:  LESSON 1

    This lesson is the first lesson in flying an aircraft. After this
lesson you will be familiar with the process of starting the plane and
taking off.  The specific skills taught in this lesson are:

        (a)  Conducting all preflight checks and inspections

        (b)  Basic communications with the tower

    There is one problem in the lesson, a comprehensive test of the
skills described above.  Good luck.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Now, we are ready to ***create problem***. This command is set up similarly to the task definition, first you verify the objects, then set the scene and goal. The difference with the ***create problem*** command is that the scene and goal default to the task's initial conditions and objectives. If an object was involved with more than one task, then the default are set assuming that the tasks listed were given in the order of application.

You will be asked to verify the cast and props for the problem. You cannot remove any cast member -- you only option is to change a cast member to a new type. The new type must be a derived object type of the one given. Therefore, if you select [1] below, then there must be a derived object of pilot loaded into the MEBuilder session. If no such types are loaded in, then you cannot change it.

We are going to do the comprehensive test first because it is actually the easier problem to create. Later on we will show you how to re-order a set of problems.

```
[LESSON:pilot training I]> create problem
Name the new problem> Comprehensive Test
The student will play the role of "captian jack".
You may modify the type of any of the objects by selecting its
index below.  The new type must be a derived object of the current
type and must already be loaded into the session.
The current set of objects are:
[1]   Character: "captain jack" of "pilot"
[2]   Prop: "the aircraft" of "aircraft"
[3]   Prop: "the huffer" of "huffer"
[4]   Prop: "the wind sock" of "wind sock"
[5]   Accept this list and continue
Choose one or more of the above or "none"> 5
The current initial setting for "you" is
  based on the initial conditions of task "prep aircraft"
The current objectives for "you" is
  based on the objectives of task "prep aircraft"
The current initial setting for "the aircraft" is
  based on the initial conditions of task "prep aircraft"
The current objectives for "the aircraft" is
  based on the objectives of task "prep aircraft"
The current initial setting for "the huffer" is
  based on the initial conditions of task "prep aircraft"
The current objectives for "the huffer" is
  based on the objectives of task "prep aircraft"
The current initial setting for "the wind sock" is
  based on the initial conditions of task "prep aircraft"
The current objectives for "the wind sock" is
  based on the objectives of task "prep aircraft"
Now working on problem "Comprehensive Test".
Type "quit" to return to Lesson Building prompt.
[PROB:pilot training I:1]>
```

The prompt has now changed again. We are in the Problem Loop. In this loop, we can adjust the initial scene and objectives, or view them, using the set scene, view scene, set goal, and view goal commands. We can also ***edit problem intro*** and ***view problem intro*** for the problem's introductory text. The ***view problem*** command is also useful here.

```
[PROB:pilot training I:1]> view problem
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Problem #1 of lesson "pilot training I"
Name of Problem>> "Comprehensive Test"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Description of Problem:
Now that you have successfully completed the various phases of the
process, let's put the whole thing together from the start.  Good
luck!

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[PROB:pilot training I:1]>
```

Because we are making a comprehensive test, the task and problem are basically the same. So for this problem we are done. The ***quit*** command will return you to the Lesson Loop.

```
[PROB:pilot training I:1]> quit
[LESSON:pilot training I]>
```

(Presently, there are a lot of enhancements for the Problem Loop that are planned but not yet implemented -- some of the enhancements will include overriding or blocking random events, changing some of the frequencies of some events, etc. etc. These are in the works.)

Next we will create the other two problems. We will have the first problem be for the First Half of the task, defined as the point where we are looking to leave the terminal (which is after the subprocedures in step 8). We will create the problem as before, but now we will invoke the ***set goal*** command for each object and tell it the exact point in the task where we want to end.

```
[LESSON:pilot training I]> create problem named First Half
The student will play the role of "captain jack".
You may modify the type of any of the objects by selecting its
index below.  The new type must be a derived object of the current
type and must already be loaded into the session.
Type "none" for no changes.
The current set of objects are:
[1]   Character: "captain jack" of "pilot"
[2]   Prop: "B-10" of "aircraft"
[3]   Prop: "the huffer" of "huffer"
[4]   Prop: "the wind sock" of "wind sock"
[5]   Accept the list and continue
Choose one of the above> 5
This new problem is number 2.
........etc..............
[PROB:pilot training I:2]> set goal for captain jack
This is the current scenario for the object:
[1]   captain jack's flight clearance is immaterial
[2]   captain jack's taxi clearance is immaterial
[3]   captain jack is cleared for takeoff
Your options are:
  step   -- Set the objectives of the object to that of a given step
            in a task.
  modify -- Make adjustments to the current objectives of the object
  start over -- Undo all changes in this command
  help   -- Prints out this message
  quit   -- queries to save changes and exits
SET GOAL>
```

You are now in a special loop for the ***set goal*** command (***set scene*** has a similar loop). The option you will normally provide is the step option, demonstrated below:

```
SET GOAL> step
The following are your choices:
[1]   Leave it alone
[2]   As it looks after step 1 of task prep aircraft
[3]   As it looks after step 2 of task prep aircraft
[4]   As it looks after step 3 of task prep aircraft
[5]   As it looks after step 4 of task prep aircraft
[6]   As it looks after step 5 of task prep aircraft
[7]   As it looks after step 6 of task prep aircraft
```

```
[8]   As it looks after step 7 of task prep aircraft
[9]   As it looks after the subprocedures in 8 of task prep aircraft
[10]  As it looks after step 9 of task prep aircraft
[11]  As it looks after step 10 of task prep aircraft
[12]  As it looks after the subprocedures in 11 of task prep aircraft
[13]  As it looks after step 12 of task prep aircraft
[14]  As it looks after step 13 of task prep aircraft
[15]  As it looks after step 14 of task prep aircraft
[16]  As it looks at the beginning of task prep aircraft
Choose one of the above> 9
The current objectives for the object is:
[1]   captain jack is cleared to depart
[2]   captain jack's taxi clearance is immaterial
[3]   captain jack's takeoff clearance is immaterial
SET GOAL> quit
```

Notice what the objectives look like. Everything is treated as immaterial except for the last change to the object made in the task. This means that the only objective that will be shown to the student is that the student must be cleared to depart. If you want something else to be shown to the student, then you can select the modify option to make adjustments -- but be very careful when doing so otherwise you may make the problem unsolvable.

After adjusting the pilot, we perform the same adjustments to the other three objects. The B-10 aircraft is demonstrated:

```
[PROB:pilot training I:2]> set goal for B-10
This is the current scenario for the object:
[1]   B-10's preflight completion is immaterial
[2]   B-10 is airborne
[3]   B-10's APU's switch position is immaterial
[4]   B-10's APU's generator's switch position is immaterial
[5]   B-10's APU's bleed air's switch position is immaterial
[6]   B-10's brakes' check status are immaterial
[7]   B-10's engine's running status are immaterial
[8]   B-10's external power's usage is immaterial
[9]   B-10's external power's switch position is immaterial
[10]  B-10's NWS's check status are immaterial
[11]  B-10's throttles' position is immaterial
[12]  B-10's trim's check status are immaterial
Your options are:
  step   -- Set the objectives of the object to that of a given step
            in a task.
  modify -- Make adjustments to the current objectives of the object
  start over -- Undo all changes in this command
  help   -- Prints out this message
  quit   -- queries to save changes and exits
SET GOAL> step
The following are your choices:
[1]   Leave it alone
[2]   As it looks after step 1 of task prep aircraft
[3]   As it looks after step 2 of task prep aircraft
[4]   As it looks after step 3 of task prep aircraft
[5]   As it looks after step 4 of task prep aircraft
[6]   As it looks after step 5 of task prep aircraft
[7]   As it looks after step 6 of task prep aircraft
[8]   As it looks after step 7 of task prep aircraft
[9]   As it looks after the subprocedures in 8 of task prep aircraft
[10]  As it looks after step 9 of task prep aircraft
[11]  As it looks after step 10 of task prep aircraft
```

```
[12]  As it looks after the subprocedures in 11 of task prep aircraft
[13]  As it looks after step 12 of task prep aircraft
[14]  As it looks after step 13 of task prep aircraft
[15]  As it looks after step 14 of task prep aircraft
[16]  As it looks at the beginning of task prep aircraft
Choose one of the above> 9
The current objectives for the object is:
[1]   B-10's preflight completion is immaterial
[2]   B-10's location is immaterial
[3]   B-10's APU's switch position is immaterial
[4]   B-10's APU's generator's switch position is immaterial
[5]   B-10's APU's bleed air's switch position is immaterial
[6]   B-10's brakes are checked
[7]   B-10's engine's running status are immaterial
[8]   B-10's external power's usage is immaterial
[9]   B-10's external power is off
[10]  B-10's NWS is checked
[11]  B-10's throttles' position is immaterial
[12]  B-10's trim's check status are immaterial
SET GOAL> quit
Save changes to objectives? yes
Objective modification completed.
Objectives set for "B-10"
```

Important: It is not necessary but strongly recommended that the same step be selected for all objects if possible. Failing to do so could have adverse consequences (there is presently no way to do this automatically).

We will skip the rest of the creation of problem 2 and all of problem 3. Problem 3 is identical in concept except that the ***set scene*** command is used to put the initial scenario for all objects to the end of step 8 in the task which the goals are left alone. Below are the intros for these problems.

```
[PROB:pilot training I:2]> view problem intro
The following is the introduction text for this lesson:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Terminal Preflight Operations

    First, we will train you on the procedure for handling the
aircraft when you first arrive.  Your job is to start with an
aircraft with everything turned off and take it through the
initial sequence of checks and gain clearance to depart the
terminal.  Good luck, Captain!

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[PROB:pilot training I:2]> quit
[LESSON:pilot training I]> work on problem number 3
Now working on problem "Second Half"
[PROB:pilot training I:3]> view problem intro
The following is the introduction text for this lesson:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Taxi and Takeoff Procedures

    In this problem, your plane is now ready to leave the terminal
and tower has granted you permission to depart.  Your job is to
taxi the aircraft to the runway, perform the last set of checks,
and fly your B-10 aircraft.  Don't forget to communicate with the
tower.  Good luck, Captain!

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
[PROB:pilot training I:3]> quit
```

Here is how our lesson looks with the ***view lesson*** command.

```
[LESSON:pilot training I]> view lesson
Lesson "pilot training I":
Cast:
captain jack. . . . . . . . . . . . . . pilot
Props:
B-10. . . . . . . . . . . . . . . . . . aircraft
the huffer. . . . . . . . . . . . . . . huffer
the wind sock . . . . . . . . . . . . . wind sock
Problem Set:
[1]   Comprehensive Test
[2]   First Half
[3]   Second Half
[LESSON:pilot training I]>
```

Obviously, the problems are not in the correct order. We now wish to order them correctly. The ***order problems*** command is used for this purpose.

```
[LESSON:pilot training I]> order problems
Choose the reordering of the problems.  Your input must be an exact
permutation of the numbers of the left of each entry.
[1]   Comprehensive Test
[2]   First Half
[3]   Second Half
Name the new order> or  2 3 1
[LESSON:pilot training I]>
```

You can perform another ***view lesson*** to see the reordering.

Now, once we have completed the problems, we are ready to ***compile lesson***. This command will assemble an METutor-ready file which, in the next section, will be demonstrated. The sequence of events of this command are -- check the integrity of all the objects, tasks, and the lesson -- then produce the METutor database. The integrity checks can be performed ahead of time using the ***check object***, ***check task***, and ***check lesson*** commands. Here is how it looks (note -- on a SPARCStation10, this took about 10 seconds to do).

```
[LESSON:pilot training I]> compile lesson
Checking integrity of object "wind sock"....OK.
Checking integrity of object "aircraft"....OK.
Checking integrity of object "aircraft trim"....OK.
Checking integrity of object "aircraft throttles"....OK.
Checking integrity of object "aircraft NWS"....OK.
Checking integrity of object "aircraft external power"....OK.
Checking integrity of object "aircraft engine"....OK.
Checking integrity of object "aircraft brakes"....OK.
Checking integrity of object "aircraft APU"....OK.
Checking integrity of object "aircraft APU bleed air"....OK.
Checking integrity of object "aircraft APU generator"....OK.
Checking integrity of object "huffer"....OK.
Checking integrity of object "pilot"....OK.
Checking integrity of task "prep aircraft"...OK.
Translating "recommended_t" Facts .....OK
Translating "precondition_t" Facts .....OK
Translating "deletepostcondition_t" Facts .....OK
Translating "addpostcondition_t" Facts .....OK
```

```
Translating "randchange_t" Facts .....OK
Translating "singular_t" Facts .....OK
Translating "plural_t" Facts .....OK
Translating "apply_text_t" Facts .....OK
Translating "delete_text_t" Facts .....OK
Translating "add_text_t" Facts .....OK
Compilation successful and marked current.
[LESSON:pilot training I]>
```

To save an METutor compiled file -- use the ***save compiled lesson*** command. The compiled file will be the name of the lesson with underscores attached with an .met extension (pilot_training_I.met). The METutor will be in the current working directory, not the library directory!!!!!

The remainder of this appendix shows the beginnings of an METutor session run from within MEBuilder using the ***run lesson*** command. All we are doing here is showing that the lesson compiled properly and that the workbook structure translated correctly. See the next section for a detailed execution of the lesson.

```
[LESSON:pilot training I]> run lesson
Running.......
+-------------------------------------------------------------+
|  Means-Ends Tutoring System -- Version 29      (METutor)    |
+-------------------------------------------------------------+
|    by Professor Rowe and CPT Galvin,  Naval PG School       |
+-------------------------------------------------------------+

Welcome.  The name of this lesson is "pilot training I".
---------------------------------------
Welcome to Pilot Training, Part I                    Takeoff


    The purpose of this lesson is to acquaint you with the
basic procedures in taking off -- including preflight checks,
tower communications, and takeoff procedures.

    There are two major segments of the process -- those
procedures that must be done at the terminal and those that
are done during taxi and takeoff.  The first two problems
will train you on the two parts; the third will bring it all
together for you for a comprehensive test.

    Good Luck, Captain!

---------------------------------------
There are 3 problems in the lesson.
You may "list" the problems, "view" a summary of a problem,
or "do" a problem.  "help" is also available.
METutor> do    list
The following are the available problems in the lesson:
[1]   First Half
[2]   Second Half
[3]   Comprehensive Test
METutor> do problem 1
Loading and checking the problem.....please wait....Done.
PROBLEM #1
----------
Terminal Preflight Operations

    First, we will train you on the procedure for handling the
```

```
aircraft when you first arrive.  Your job is to start with an
aircraft with everything turned off and take it through the
initial sequence of checks and gain clearance to depart the
terminal.  Good luck, Captain!


The following are your objectives:
you must be cleared to depart,  B-10's brakes must be checked, B-10's
      external power must be off,  B-10's NWS must be checked,  and the
      huffer must be disengaged

The following is the current situation:
B-10 is at the terminal,  the huffer is disengaged,  B-10's APU is off,
      B-10's engine is off,  B-10's throttles are off,  the huffer is
      present,  B-10's NWS is unchecked,  B-10's brakes are unchecked,
      B-10's trim is unchecked,  B-10's external power is available,  B-10's
      APU's generator is off,  B-10's external power is off,  and B-10's
      APU's bleed air is off

What do you want to do? quit
METutor> do problem 2
Loading and checking the problem.....please wait....Done.
PROBLEM #2
----------
Taxi and Takeoff Procedures

    In this problem, your plane is now ready to leave the terminal
and tower has granted you permission to depart.  Your job is to
taxi the aircraft to the runway, perform the last set of checks,
and fly your B-10 aircraft.  Don't forget to communicate with the
tower.  Good luck, Captain!


The following are your objectives:
you must be cleared for takeoff,  B-10 must be airborne,  the huffer must be
      disengaged,   and the wind sock must be checked

The following is the current situation:
B-10 is at the terminal,  you are cleared to depart,  B-10 is
      preflight inspected,  B-10's NWS is checked,  B-10's brakes are
      checked,  the huffer is disengaged,  B-10's throttles are off,  B-10's
      APU is on,  the huffer is present,  B-10's engine is running,  B-10's
      trim is unchecked,  B-10's external power is available, B-10's
      external power is off,  B-10's APU's generator is on,  and B-10's
      APU's bleed air is on

What do you want to do? quit
METutor> do problem 3
Loading and checking the problem.....please wait....Done.
PROBLEM #3
----------
Comprehensive Test

    Now that you have completed both phases of the process,
let's put the whole thing together.  Good luck!

The following are your objectives:
you must be cleared for takeoff,  B-10 must be airborne,  the huffer must be
      disengaged,   and the wind sock must be checked

The following is the current situation:
```

```
B-10 is at the terminal,  the huffer is disengaged,  B-10's APU is off,
      B-10's engine is off,  B-10's throttles are off,  the huffer is
      present,  B-10's NWS is unchecked,  B-10's brakes are unchecked,
      B-10's trim is unchecked,  B-10's external power is available,  B-10's
      APU's generator is off,  B-10's external power is off,  and B-10's
      APU's bleed air is off

What do you want to do? quit
METutor> quit

Returned to MEBuilder...
[LESSON:pilot training I]>
```

TAB 5. RUNNING THE LESSON IN METUTOR

The following is a sample run of the lesson in METutor. This sample run demonstrates several of the commands and features of METutor. The METutor session is best run from the original working directory.

Key item to notice -- remember the "have pilot request <x> clearance" operations? Notice that because the student is serving in the role of the pilot that the "have pilot" portion is chopped off and that all references to the pilot are in second person form in order to put the student more into the scenario.

```
> ~galvint/mebuild/METutor
Name the lesson file> pilot_training_I.met
+-----------------------------------------------------------+
|  Means-Ends Tutoring System -- Version 29      (METutor)  |
+-----------------------------------------------------------+
|    by Professor Rowe and CPT Galvin,  Naval PG School     |
+-----------------------------------------------------------+

Welcome.  The name of this lesson is "pilot training I".
---------------------------------------
Welcome to Pilot Training, Part I                    Takeoff


    The purpose of this lesson is to acquaint you with the
basic procedures in taking off -- including preflight checks,
tower communications, and takeoff procedures.

    There are two major segments of the process -- those
procedures that must be done at the terminal and those that
are done during taxi and takeoff.  The first two problems
will train you on the two parts; the third will bring it all
together for you for a comprehensive test.

    Good Luck, Captain!

---------------------------------------
There are 3 problems in the lesson.
You may "list" the problems, "view" a summary of a problem,
or "do" a problem.  "help" is also available.
METutor> do problem 1
Loading and checking the problem.....please wait....Done.
PROBLEM #1
----------
Terminal Preflight Operations

    First, we will train you on the procedure for handling the
aircraft when you first arrive.  Your job is to start with an
aircraft with everything turned off and take it through the
initial sequence of checks and gain clearance to depart the
terminal.  Good luck, Captain!


The following are your objectives:
you must be cleared to depart,  B-10's brakes must be checked, B-10's
      external power must be off,  B-10's NWS must be checked,  and the
      huffer must be disengaged

The following is the current situation:
B-10 is at the terminal,  the huffer is disengaged,  B-10's APU is off,
```

```
      B-10's engine is off,  B-10's throttles are off,  the huffer is
      present,  B-10's NWS is unchecked,  B-10's brakes are unchecked,
      B-10's trim is unchecked,  B-10's external power is available,  B-10's
      APU's generator is off,  B-10's external power is off,  and B-10's
      APU's bleed air is off

What do you want to do? help
------------------------------------------------------------
You may enter an operator or one of these special commands:
   help            -- print this text
   quit            -- return to METutor main level
   view state      -- pretty prints the current state
   view objectives -- pretty prints your objectives
   query operator <operator>
      -- prints all information about an operator.
   query object   <object>
      -- prints operators available on an object.
   query fact     <object>
      -- prints all information about a fact or objective.
The following are the operators available in this lesson:
[1]   adjust B-10's trim
[2]   check B-10's NWS
[3]   check B-10's brakes
[4]   check the wind sock
[5]   conduct preflight inspection on B-10
[6]   disengage B-10's external power
[7]   disengage the huffer
[8]   engage B-10's APU's bleed air
[9]   engage B-10's APU's generator
[10]  engage B-10's external power
[11]  engage the huffer
[12]  fly the B-10
[13]  max B-10's throttles
[14]  request flight clearance
[15]  request takeoff clearance
[16]  request taxi clearance
[17]  shut off B-10's APU
[18]  start B-10's APU
[19]  start B-10's engine
[20]  taxi B-10
------------------------------------------------------------
What do you want to do? conduct preflight inspection on B-10
You chose to conduct preflight inspection on B-10.

OK.
[1]   B-10 is now preflight inspected

What do you want to do? engage B-10's external power
You chose to engage B-10's external power.

OK.
[1]   B-10's external power is now on

What do you want to do? engage the huffer
You chose to engage the huffer.

OK.
[1]   the huffer is now engaged

What do you want to do? request takeoff clearance
You chose to request takeoff clearance.
```

```
That action requires that:
[1]   you must be cleared for taxi
[2]   B-10 must be on the runway

What do you want to do? taxi B-10
You chose to taxi B-10.

That action requires that:
[1]   you must be cleared for taxi

What do you want to do? request taxi clearance
You chose to request taxi clearance.

That action requires that:
[1]   you must be cleared to depart
[2]   B-10's brakes must be checked

What do you want to do? view objectives
The following are your objectives:
you must be cleared to depart,  B-10's brakes must be checked, B-10's
      external power must be off,  B-10's NWS must be checked,  and the
      huffer must be disengaged
What do you want to do? query fact B-10's external power is on

The following operators are recommended for achieving this fact:
[1]   engage B-10's external power

What do you want to do? query object the huffer

The following can be performed on "the huffer".
** Operator = "engage the huffer":
   The operator is intended to achieve "the huffer would be engaged ".

** Operator = "disengage the huffer":
   The operator is intended to achieve "the huffer would be disengaged ".

What do you want to do? query operation check the wind sock
Sorry, that is not a valid command.  Please try again.
What do you want to do? query operator check the wind c sockk
The following is true about "check the wind sock":
** The operator is recommended for achieving "the wind sock is checked "
** The operator is recommended for achieving "the wind sock is checked "
** The precondition for the operator is "B-10 must be on the runway  and the
      wind sock must not be checked ".
** The postcondition for the operator is "" while "the wind sock would be
      checked ".
** The postcondition for the operator is "" while "the wind sock would be
      checked ".
** The postcondition for the operator is "" while "the wind sock would be
      checked ".
** The postcondition for the operator is "" while "the wind sock would be
      checked ".
What do you want to do? quit
METutor> quit
>
```

APPENDIX D. SAMPLE DATA FILES

This appendix contains examples of data files produced by MEBuilder during the session scripted in Appendix C. Then following are the files included in this Appendix:

Tab 1. mebuild.lib -- The library directory file produced by MEBuildLIB

Tab 2. pilot.cls -- The definition file for the "pilot" object.

Tab 3. aircraft.cls -- The definition file for the "aircraft" object.

Tab 4. prep_aircraft.tsk -- The definition file for the "prep aircraft" task.

Tab 5. pilot_training_I.les -- The definition file for the "pilot training I" lesson.

Tab 6. pilot_training_I.met -- The compiled METutor file. Appendix E contains excerpts of a script run of this file in METutor.

TAB 1. LIBRARY DIRECTORY FILE

```
/*****************************************************************/
/* MEBuilder Library File -- Directory of Classes and Lessons */
/*****************************************************************/


:- dynamic      type_of_prolog_file/1.

:- multifile    type_of_prolog_file/1.

:- dynamic      library_class_entry/4, library_task_entry/4,
                library_lesson_entry/5, library_link/1.

:- multifile    library_class_entry/4, library_task_entry/4,
                library_lesson_entry/5, library_link/1.


type_of_prolog_file('MEBuilder Library Directory File').


/* library_class_entry/4 */
library_class_entry([aircraft,trim],'./lib/aircraft_trim.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,throttles],'./lib/aircraft_throttle
.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,'NWS'],'./lib/aircraft_NWS.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,external,power],'./lib/aircraft_ext
rnal_power.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,engine],'./lib/aircraft_engine.cls'
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,brakes],'./lib/aircraft_brakes.cls'
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,'APU',bleed,air],'./lib/aircraft_AP
_bleed_air.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry([aircraft,'APU',generator],'./lib/aircraft_AP
_generator.cls',
    date(94,6,26,10,56,35),
    [prop]).
library_class_entry(pilot,'./lib/pilot.cls',
    date(94,6,26,11,6,34),
    [character]).
library_class_entry([wind,sock],'./lib/wind_sock.cls',
    date(94,6,26,11,6,39),
    [prop]).
library_class_entry(huffer,'./lib/huffer.cls',
    date(94,6,26,11,9,7),
    [prop]).
```

```
library_class_entry([aircraft,'APU'],'./lib/aircraft_APU.cls',
    date(94,6,26,13,54,32),
    [[aircraft,'APU',bleed,air],
     [aircraft,'APU',generator],
     prop]).
library_class_entry(aircraft,'./lib/aircraft.cls',
    date(94,6,26,15,16,54),
    [[aircraft,trim],
     [aircraft,throttles],
     [aircraft,'NWS'],
     [aircraft,external,power],
     [aircraft,engine],
     [aircraft,brakes],
     [aircraft,'APU'],
     prop]).


/* library_task_entry/4 */
library_task_entry([prep,aircraft],'./lib/prep_aircraft.tsk',
    date(94,6,27,10,48,4),
    [[wind,sock],
     aircraft,
     [aircraft,trim],
     [aircraft,throttles],
     [aircraft,'NWS'],
     [aircraft,external,power],
     [aircraft,engine],
     [aircraft,brakes],
     [aircraft,'APU'],
     [aircraft,'APU',bleed,air],
     [aircraft,'APU',generator],
     huffer,
     pilot]).


/* library_lesson_entry/5 */
library_lesson_entry([pilot,training,'I'],'./lib/pilot_training_I.les',
    date(94,6,27,11,55,58),
    [[wind,sock],
     huffer,
     aircraft,
     pilot],
    [[prep,aircraft]]).


/* library_link/1 */
```

TAB 2. OBJECT DEFINITION FILE FOR PILOT

```
/*****************************************************************/
/*              MEBuilder Class Definition File                  */
/*****************************************************************/


:- dynamic      type_of_prolog_file/1.

:- multifile    type_of_prolog_file/1.

:- dynamic      class_def/2, component/4, property_set/4,
                property_display_data/4, relation/4, daemon/6,
                operation/8, operation_display_data/4.


:- multifile    class_def/2, component/4, property_set/4,
                property_display_data/4, relation/4, daemon/6,
                operation/8, operation_display_data/4.


type_of_prolog_file('MEBuilder Library Class Definition File').


/* class_def/2 */
class_def(pilot,[character]).


/* component/4 */


/* property_set/4 */
property_set(pilot,[flight,clearance],
    oneof(['cleared to depart']),
    not_hideable).
property_set(pilot,[taxi,clearance],
    oneof(['cleared for taxi']),
    not_hideable).
property_set(pilot,[takeoff,clearance],
    oneof(['cleared for takeoff']),
    not_hideable).


/* property_display_data/4 */


/* relation/4 */


/* daemon/6 */


/* operation/8 */
operation(pilot,[aircraft],
    have,pilot,[request,flight,clearance],
    [[],
     [on('APU''s',bleed,air)]],
    'cleared to depart',
    [[],
```

```
      []]).
operation(pilot,[aircraft],
    have,pilot,[request,taxi,clearance],
    [['cleared to depart'],
     [checked(brakes)]],
    'cleared for taxi',
    [[],
     []]).
operation(pilot,[aircraft],
    have,pilot,[request,takeoff,clearance],
    [['cleared for taxi'],
     ['on the runway']],
    'cleared for takeoff',
    [[],
     []]).


/* operation_display_data/4 */
```

TAB 3. OBJECT DEFINITION FILE FOR AIRCRAFT

```
/*****************************************************************/
/*              MEBuilder Class Definition File                  */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.

:- dynamic     class_def/2, component/4, property_set/4,
               property_display_data/4, relation/4, daemon/6,
               operation/8, operation_display_data/4.

:- multifile   class_def/2, component/4, property_set/4,
               property_display_data/4, relation/4, daemon/6,
               operation/8, operation_display_data/4.


type_of_prolog_file('MEBuilder Library Class Definition File').


/* class_def/2 */
class_def(aircraft,[prop]).


/* component/4 */
component(aircraft,[aircraft,'APU'],'APU',
    default).
component(aircraft,[aircraft,brakes],brakes,
    default).
component(aircraft,[aircraft,engine],engine,
    default).
component(aircraft,[aircraft,external,power],[external,power],
    default).
component(aircraft,[aircraft,'NWS'],'NWS',
    default).
component(aircraft,[aircraft,throttles],throttles,
    default).
component(aircraft,[aircraft,trim],trim,
    default).


/* property_set/4 */
property_set(aircraft,[preflight,completion],
    oneof(['preflight inspected']),
    not_hideable).
property_set(aircraft,location,
    oneof(['at the terminal','on the runway',airborne]),
    not_hideable).


/* property_display_data/4 */


/* relation/4 */


/* daemon/6 */
```

```
/* operation/8 */
operation(aircraft,[pilot,huffer],
    check,'NWS',[],
    [[off(external,power)],
     []],
    checked('NWS'),
    [[],
     [],
     []]).
operation(aircraft,[pilot],
    check,brakes,[],
    [[checked('NWS')],
     []],
    checked(brakes),
    [[],
     []]).
operation(aircraft,[pilot,[wind,sock]],
    adjust,trim,[],
    [[],
     [],
     [checked]],
    checked(trim),
    [[],
     [],
     []]).
operation(aircraft,[pilot],
    [shut,off],'APU',[],
    [[checked(trim)],
     ['cleared for takeoff']],
    off('APU'),
    [[],
     []]).
operation(aircraft,[pilot],
    taxi,aircraft,[],
    [[],
     ['cleared for taxi']],
    'on the runway',
    [[],
     []]).
operation(aircraft,[pilot],
    max,throttles,[],
    [[off('APU')],
     ['cleared for takeoff']],
    full(throttles),
    [[],
     []]).
operation(aircraft,[pilot],
    [fly,the],aircraft,[],
    [[full(throttles)],
     []],
    airborne,
    [[],
     []]).
operation(aircraft,[pilot],
    start,'APU',[],
    [['preflight inspected',running(engine)],
     []],
    on('APU'),
    [[],
```

```
     []]).
operation(aircraft,[pilot],
    [conduct,preflight,inspection,on],aircraft,[],
    [['at the terminal'],
     []],
    'preflight inspected',
    [[],
     []]).
operation(aircraft,[pilot],
    engage,[external,power],[],
    [['preflight inspected'],
     []],
    on(external,power),
    [[],
     []]).
operation(aircraft,[huffer,pilot],
    start,engine,[],
    [['preflight inspected'],
     [engaged],
     []],
    running(engine),
    [[],
     [],
     []]).
operation(aircraft,[huffer,pilot],
    disengage,[external,power],[],
    [[],
     [disengaged],
     []],
    off(external,power),
    [[],
     [],
     []]).


/* operation_display_data/4 */
```

TAB 4. TASK DEFINITION FILE FOR PREP_AIRCRAFT

```
/*****************************************************************/
/*                MEBuilder Task Definition File                 */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.

:- dynamic     task/3, initial_conditions/2, objectives/2, stage/6,
               action/6, unordered_action/2, relation/3.

:- multifile   task/3, initial_conditions/2, objectives/2, stage/6,
               action/6, unordered_action/2, relation/3.


type_of_prolog_file('MEBuilder Library Task Definition File').


/* task/3 */
task([prep,aircraft],[pilot,pilot],
    [[huffer,huffer],
     [aircraft,aircraft],
     [[wind,sock],[wind,sock]]]).


/* initial_conditions/2 */
initial_conditions(pilot,
    [not('cleared to depart'(pilot)),
     not('cleared for taxi'(pilot)),
     not('cleared for takeoff'(pilot))]).
initial_conditions(huffer,
    [present(huffer),
     disengaged(huffer)]).
initial_conditions(aircraft,
    [not('preflight inspected'(aircraft)),
     'at the terminal'(aircraft),
     off('aircraft''s','APU'),
     off('aircraft''s','APU''s',generator),
     off('aircraft''s','APU''s',bleed,air),
     unchecked('aircraft''s',brakes),
     off('aircraft''s',engine),
     available('aircraft''s',external,power),
     off('aircraft''s',external,power),
     unchecked('aircraft''s','NWS'),
     off('aircraft''s',throttles),
     unchecked('aircraft''s',trim)]).
initial_conditions([wind,sock],
    [not(checked(wind,sock))]).


/* objectives/2 */
objectives(pilot,
    ['cleared to depart'(pilot),
     'cleared for taxi'(pilot),
     'cleared for takeoff'(pilot)]).
objectives(huffer,
    [present(huffer),
```

```
    immaterial('huffer''s',engagement)]).
objectives(aircraft,
    ['preflight inspected'(aircraft),
     airborne(aircraft),
     immaterial('aircraft''s','APU''s',switch,position),
     immaterial('aircraft''s','APU''s','generator''s',switch,position),
     immaterial('aircraft''s','APU''s',bleed,'air''s',switch,position),
     checked('aircraft''s',brakes),
     running('aircraft''s',engine),
     immaterial('aircraft''s',external,'power''s',usage),
     immaterial('aircraft''s',external,'power''s',switch,position),
     checked('aircraft''s','NWS'),
     full('aircraft''s',throttles),
     checked('aircraft''s',trim)]).
objectives([wind,sock],
    [checked(wind,sock)]).


/* stage/6 */
stage(start,linear,none,linear,
    [],
    []).
stage(q1,linear,none,linear,
    [[start,101,1]],
    []).
stage(q2,linear,none,linear,
    [[q1,102,1]],
    []).
stage(q3,linear,none,linear,
    [[q2,103,1]],
    []).
stage(q4,linear,none,linear,
    [[q3,104,1]],
    []).
stage(q5,linear,none,linear,
    [[q4,105,1]],
    []).
stage(q6,linear,none,linear,
    [[q5,106,1]],
    []).
stage(q17,linear,none,joining,
    [[join1,lambda,1]],
    []).
stage(q13,linear,none,linear,
    [[q17,113,1]],
    []).
stage(q14,and-split,q15,linear,
    [[q13,114,1]],
    []).
stage(q16,linear,none,linear,
    [[q14,116,1]],
    [[q14,and,1]]).
stage(join2,linear,none,joining,
    [[q16,117,1],[q14,115,1]],
    [[q14,and,1]]).
stage(q15,linear,none,joining,
    [[join2,lambda,1]],
    []).
stage(q18,linear,none,linear,
    [[q15,118,1]],
    []).
```

```
stage(q19,linear,none,linear,
    [[q18,119,1]],
    []).
stage(done,no-actions,none,linear,
    [[q19,120,1]],
    []).
stage(join1,linear,none,joining,
    [[q20,108,1],[q11,112,1]],
    [[q20,and,1]]).
stage(q11,linear,none,linear,
    [[q10,111,1]],
    [[q20,and,1]]).
stage(q20,and-split,q17,linear,
    [[q6,107,1]],
    []).
stage(q10,linear,none,linear,
    [[q21,110,1]],
    [[q20,and,1]]).
stage(q21,linear,none,linear,
    [[q20,109,1]],
    [[q20,and,1]]).


/* action/6 */
action(101,conduct(preflight,inspection,on,aircraft),
    [],
    [],
    [],
    []).
action(102,engage('aircraft''s',external,power),
    [],
    [],
    [],
    []).
action(103,engage(huffer),
    [],
    [],
    [],
    []).
action(104,start('aircraft''s',engine),
    [],
    [],
    [],
    []).
action(105,start('aircraft''s','APU'),
    [],
    [],
    [],
    []).
action(106,engage('aircraft''s','APU''s',generator),
    [],
    [],
    [],
    []).
action(107,engage('aircraft''s','APU''s',bleed,air),
    [],
    [],
    [],
    []).
action(108,have(pilot,request,flight,clearance),
    [],
```

```
    [],
    [],
    []).
action(109,disengage(huffer),
    [],
    [],
    [],
    []).
action(110,disengage('aircraft''s',external,power),
    [],
    [],
    [],
    []).
action(111,check('aircraft''s','NWS'),
    [],
    [],
    [],
    []).
action(112,check('aircraft''s',brakes),
    [],
    [],
    [],
    []).
action(113,have(pilot,request,taxi,clearance),
    [],
    [],
    [],
    []).
action(114,taxi(aircraft),
    [],
    [],
    [],
    []).
action(115,have(pilot,request,takeoff,clearance),
    [],
    [],
    [],
    []).
action(116,check(wind,sock),
    [],
    [],
    [],
    []).
action(117,adjust('aircraft''s',trim),
    [],
    [],
    [],
    []).
action(118,shut(off,'aircraft''s','APU'),
    [],
    [],
    [],
    []).
action(119,max('aircraft''s',throttles),
    [],
    [],
    [],
    []).
action(120,fly(the,aircraft),
    [],
    [],
```

```
    [],
    []).


/* unordered_action/2 */


/* relation/3 */
```

TAB 5. LESSON DEFINITION FILE FOR PILOT_TRAINING

```
/*****************************************************************/
/*              MEBuilder Lesson Definition File                 */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.

:- dynamic     lesson/4, lesson_intro/1, problem/5, problem_intro/2,
               initial_setting/3, objectives/3,  side_effect_override/5.

:- multifile   lesson/4, lesson_intro/1, problem/5, problem_intro/2,
               initial_setting/3, objectives/3, side_effect_override/5.


type_of_prolog_file('MEBuilder Library Lesson Definition File').


/* lesson/4 */
lesson([pilot,training,'I'],
    [[[captain,jack,pilot,default]],
    [['B-10',aircraft,default],
     [[the,huffer],huffer,default],
     [[the,wind,sock],[wind,sock],default]],
    [[prep,aircraft]]).

/* lesson_intro/1 */
lesson_intro('Welcome to Pilot Training, Part I                    Takeoff').
lesson_intro('').
lesson_intro('').
lesson_intro('    The purpose of this lesson is to acquaint you with the').
lesson_intro('basic procedures in taking off -- including preflight checks,').
lesson_intro('tower communications, and takeoff procedures.').
lesson_intro('').
lesson_intro('    There are two major segments of the process -- those ').
lesson_intro('procedures that must be done at the terminal and those that').
lesson_intro('are done during taxi and takeoff.  The first two problems').
lesson_intro('will train you on the two parts; the third will bring it all').
lesson_intro('together for you for a comprehensive test.').
lesson_intro('').
lesson_intro('    Good Luck, Captain!').


/* problem/5 */
problem(2,['Second','Half'],[captain,jack],
    [[[captain,jack],pilot,default]],
    [['B-10',aircraft,default],
     [[the,huffer],huffer,default],
     [[the,wind,sock],[wind,sock],default]]).
problem(3,['Comprehensive','Test'],[captain,jack],
    [[[captain,jack],pilot,default]],
    [['B-10',aircraft,default],
     [[the,huffer],huffer,default],
     [[the,wind,sock],[wind,sock],default]]).
problem(1,['First','Half'],[captain,jack],
    [[[captain,jack],pilot,default]],
    [['B-10',aircraft,default],
```

```
     [[the,huffer],huffer,default],
     [[the,wind,sock],[wind,sock],default]]).


/* problem_intro/2 */
problem_intro(2,'Taxi and Takeoff Procedures').
problem_intro(2,'').
problem_intro(2,'    In this problem, your plane is now ready to leave the terminal').
problem_intro(2,'and tower has granted you permission to depart.  Your job is to').
problem_intro(2,'taxi the aircraft to the runway, perform the last set of checks, ').
problem_intro(2,'and fly your B-10 aircraft.  Don''t forget to communicate with the').
problem_intro(2,'tower.  Good luck, Captain!').
problem_intro(2,'').
problem_intro(3,'Comprehensive Test').
problem_intro(3,'').
problem_intro(3,'    Now that you have completed both phases of the process, ').
problem_intro(3,'let''s put the whole thing together.  Good luck!').
problem_intro(1,'Terminal Preflight Operations').
problem_intro(1,'').
problem_intro(1,'    First, we will train you on the procedure for handling the').
problem_intro(1,'aircraft when you first arrive.  Your job is to start with an').
problem_intro(1,'aircraft with everything turned off and take it through the').
problem_intro(1,'initial sequence of checks and gain clearance to depart the').
problem_intro(1,'terminal.  Good luck, Captain!').
problem_intro(1,'').


/* initial_setting/3 */
initial_setting(2,[captain,jack],
    ['cleared to depart'(captain,jack),
     not('cleared for taxi'(captain,jack)),
     not('cleared for takeoff'(captain,jack))]).
initial_setting(2,'B-10',
    [checked('B-10''s',brakes),
     off('B-10''s',external,power),
     checked('B-10''s','NWS'),
     'preflight inspected'('B-10'),
     'at the terminal'('B-10'),
     on('B-10''s','APU'),
     on('B-10''s','APU''s',generator),
     on('B-10''s','APU''s',bleed,air),
     running('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim)]).
initial_setting(2,[the,huffer],
    [disengaged(the,huffer),
     present(the,huffer)]).
initial_setting(2,[the,wind,sock],
    [not(checked(the,wind,sock))]).
initial_setting(3,[captain,jack],
    [not('cleared to depart'(captain,jack)),
     not('cleared for taxi'(captain,jack)),
     not('cleared for takeoff'(captain,jack))]).
initial_setting(3,'B-10',
    [not('preflight inspected'('B-10')),
     'at the terminal'('B-10'),
     off('B-10''s','APU'),
     off('B-10''s','APU''s',generator),
     off('B-10''s','APU''s',bleed,air),
     unchecked('B-10''s',brakes),
```

```
     off('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',external,power),
     unchecked('B-10''s','NWS'),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim)]).
initial_setting(3,[the,huffer],
    [present(the,huffer),
     disengaged(the,huffer)]).
initial_setting(3,[the,wind,sock],
    [not(checked(the,wind,sock))]).
initial_setting(1,[captain,jack],
    [not('cleared to depart'(captain,jack)),
     not('cleared for taxi'(captain,jack)),
     not('cleared for takeoff'(captain,jack))]).
initial_setting(1,'B-10',
    [not('preflight inspected'('B-10')),
     'at the terminal'('B-10'),
     off('B-10''s','APU'),
     off('B-10''s','APU''s',generator),
     off('B-10''s','APU''s',bleed,air),
     unchecked('B-10''s',brakes),
     off('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',external,power),
     unchecked('B-10''s','NWS'),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim)]).
initial_setting(1,[the,huffer],
    [present(the,huffer),
     disengaged(the,huffer)]).
initial_setting(1,[the,wind,sock],
    [not(checked(the,wind,sock))]).


/* objectives/3 */
objectives(2,[captain,jack],
    [immaterial(captain,'jack''s',flight,clearance),
     immaterial(captain,'jack''s',taxi,clearance),
     'cleared for takeoff'(captain,jack)]).
objectives(2,'B-10',
    [immaterial('B-10''s',preflight,completion),
     airborne('B-10'),
     immaterial('B-10''s','APU''s',switch,position),
     immaterial('B-10''s','APU''s','generator''s',switch,position),
     immaterial('B-10''s','APU''s',bleed,'air''s',switch,position),
     immaterial('B-10''s','brakes''',check,status),
     immaterial('B-10''s','engine''s',running,status),
     immaterial('B-10''s',external,'power''s',usage),
     immaterial('B-10''s',external,'power''s',switch,position),
     immaterial('B-10''s','NWS''s',check,status),
     immaterial('B-10''s','throttles''',position),
     immaterial('B-10''s','trim''s',check,status)]).
objectives(2,[the,huffer],
    [immaterial(the,'huffer''s',presence),
     disengaged(the,huffer)]).
objectives(2,[the,wind,sock],
    [checked(the,wind,sock)]).
objectives(3,[captain,jack],
    [immaterial(captain,'jack''s',flight,clearance),
     immaterial(captain,'jack''s',taxi,clearance),
```

```
     'cleared for takeoff'(captain,jack)]).
objectives(3,'B-10',
    [immaterial('B-10''s',preflight,completion),
     airborne('B-10'),
     immaterial('B-10''s','APU''s',switch,position),
     immaterial('B-10''s','APU''s','generator''s',switch,position),
     immaterial('B-10''s','APU''s',bleed,'air''s',switch,position),
     immaterial('B-10''s','brakes''',check,status),
     immaterial('B-10''s','engine''s',running,status),
     immaterial('B-10''s',external,'power''s',usage),
     immaterial('B-10''s',external,'power''s',switch,position),
     immaterial('B-10''s','NWS''s',check,status),
     immaterial('B-10''s','throttles''',position),
     immaterial('B-10''s','trim''s',check,status)]).
objectives(3,[the,huffer],
    [immaterial(the,'huffer''s',presence),
     disengaged(the,huffer)]).
objectives(3,[the,wind,sock],
    [checked(the,wind,sock)]).
objectives(1,[captain,jack],
    ['cleared to depart'(captain,jack),
     immaterial(captain,'jack''s',taxi,clearance),
     immaterial(captain,'jack''s',takeoff,clearance)]).
objectives(1,'B-10',
    [immaterial('B-10''s',preflight,completion),
     immaterial('B-10''s',location),
     immaterial('B-10''s','APU''s',switch,position),
     immaterial('B-10''s','APU''s','generator''s',switch,position),
     immaterial('B-10''s','APU''s',bleed,'air''s',switch,position),
     checked('B-10''s',brakes),
     immaterial('B-10''s','engine''s',running,status),
     immaterial('B-10''s',external,'power''s',usage),
     off('B-10''s',external,power),
     checked('B-10''s','NWS'),
     immaterial('B-10''s','throttles''',position),
     immaterial('B-10''s','trim''s',check,status)]).
objectives(1,[the,huffer],
    [immaterial(the,'huffer''s',presence),
     disengaged(the,huffer)]).
objectives(1,[the,wind,sock],
    [immaterial(the,wind,'sock''s',check,status)]).


/* side_effect_override/5 */
```

TAB 6. METUTOR READY FILE FOR PILOT TRAINING

```
/*****************************************************************/
/*  Means-Ends Lesson Definition File -- Runnable in METutor  */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.

:- dynamic     lesson/1, lesson_intro/1, problem/3, problem_intro/2,
               problem_domain/3, start_state_t/3, goal_t/3, recommended_t/4,
               precondition_t/5, deletepostcondition_t/5,
               addpostcondition_t/5, randchange_t/7, singular_t/2,
               plural_t/2, apply_text_t/4, delete_text_t/4, add_text_t/4.

:- multifile   lesson/1, lesson_intro/1, problem/3, problem_intro/2,
               problem_domain/3, start_state_t/3, goal_t/3, recommended_t/4,
               precondition_t/5, deletepostcondition_t/5,
               addpostcondition_t/5, randchange_t/7, singular_t/2,
               plural_t/2, apply_text_t/4, delete_text_t/4, add_text_t/4.


type_of_prolog_file('METutor Lesson File').


/* lesson/1 */
lesson([pilot,training,'I']).


/* lesson_intro/1 */
lesson_intro('Welcome to Pilot Training, Part I                    Takeoff').
lesson_intro('').
lesson_intro('').
lesson_intro('    The purpose of this lesson is to acquaint you with the').
lesson_intro('basic procedures in taking off -- including preflight checks,').
lesson_intro('tower communications, and takeoff procedures.').
lesson_intro('').
lesson_intro('    There are two major segments of the process -- those ').
lesson_intro('procedures that must be done at the terminal and those that').
lesson_intro('are done during taxi and takeoff.  The first two problems').
lesson_intro('will train you on the two parts; the third will bring it all').
lesson_intro('together for you for a comprehensive test.').
lesson_intro('').
lesson_intro('    Good Luck, Captain!').


/* problem/3 */
problem(1,['First','Half'],[captain,jack]).
problem(2,['Second','Half'],[captain,jack]).
problem(3,['Comprehensive','Test'],[captain,jack]).


/* problem_intro/2 */
problem_intro(1,'Terminal Preflight Operations').
problem_intro(1,'').
problem_intro(1,'    First, we will train you on the procedure for handling the').
problem_intro(1,'aircraft when you first arrive.  Your job is to start with an').
problem_intro(1,'aircraft with everything turned off and take it through the').
```

```
problem_intro(1,'initial sequence of checks and gain clearance to depart the').
problem_intro(1,'terminal.  Good luck, Captain!').
problem_intro(1,'').
problem_intro(2,'Taxi and Takeoff Procedures').
problem_intro(2,'').
problem_intro(2,'    In this problem, your plane is now ready to leave the terminal').
problem_intro(2,'and tower has granted you permission to depart.  Your job is to').
problem_intro(2,'taxi the aircraft to the runway, perform the last set of checks, ').
problem_intro(2,'and fly your B-10 aircraft.  Don''t forget to communicate with the').
problem_intro(2,'tower.  Good luck, Captain!').
problem_intro(2,'').
problem_intro(3,'Comprehensive Test').
problem_intro(3,'').
problem_intro(3,'    Now that you have completed both phases of the process, ').
problem_intro(3,'let''s put the whole thing together.  Good luck!').


/* problem_domain/3 */
problem_domain(1,pilot,
    [[captain,jack]]).
problem_domain(1,aircraft,
    ['B-10']).
problem_domain(1,huffer,
    [[the,huffer]]).
problem_domain(1,[wind,sock],
    [[the,wind,sock]]).
problem_domain(2,pilot,
    [[captain,jack]]).
problem_domain(2,aircraft,
    ['B-10']).
problem_domain(2,huffer,
    [[the,huffer]]).
problem_domain(2,[wind,sock],
    [[the,wind,sock]]).
problem_domain(3,pilot,
    [[captain,jack]]).
problem_domain(3,aircraft,
    ['B-10']).
problem_domain(3,huffer,
    [[the,huffer]]).
problem_domain(3,[wind,sock],
    [[the,wind,sock]]).


/* start_state_t/3 */
start_state_t(1,
    [],
    ['at the terminal'('B-10'),
     off('B-10''s','APU'),
     off('B-10''s','APU''s',generator),
     off('B-10''s','APU''s',bleed,air),
     unchecked('B-10''s',brakes),
     off('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',external,power),
     unchecked('B-10''s','NWS'),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim),
     present(the,huffer),
     disengaged(the,huffer)]).
start_state_t(2,
```

```
    [],
    ['cleared to depart'(you),
     checked('B-10''s',brakes),
     off('B-10''s',external,power),
     checked('B-10''s','NWS'),
     'preflight inspected'('B-10'),
     'at the terminal'('B-10'),
     on('B-10''s','APU'),
     on('B-10''s','APU''s',generator),
     on('B-10''s','APU''s',bleed,air),
     running('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim),
     disengaged(the,huffer),
     present(the,huffer)]).
start_state_t(3,
    [],
    ['at the terminal'('B-10'),
     off('B-10''s','APU'),
     off('B-10''s','APU''s',generator),
     off('B-10''s','APU''s',bleed,air),
     unchecked('B-10''s',brakes),
     off('B-10''s',engine),
     available('B-10''s',external,power),
     off('B-10''s',external,power),
     unchecked('B-10''s','NWS'),
     off('B-10''s',throttles),
     unchecked('B-10''s',trim),
     present(the,huffer),
     disengaged(the,huffer)]).


/* goal_t/3 */
goal_t(1,
    [],
    ['cleared to depart'(you),
     checked('B-10''s',brakes),
     off('B-10''s',external,power),
     checked('B-10''s','NWS'),
     disengaged(the,huffer)]).
goal_t(2,
    [],
    ['cleared for takeoff'(you),
     airborne('B-10'),
     disengaged(the,huffer),
     checked(the,wind,sock)]).
goal_t(3,
    [],
    ['cleared for takeoff'(you),
     airborne('B-10'),
     disengaged(the,huffer),
     checked(the,wind,sock)]).


/* recommended_t/4 */
recommended_t([captain,jack],
    [aircraft,
     pilot],
    ['preflight inspected'(arg(1))],
    conduct(preflight,inspection,on,arg(1))).
```

```
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [on(parg(1),external,power)],
    engage(parg(1),external,power)).
recommended_t([captain,jack],
    [huffer,
     aircraft],
    [engaged(arg(1))],
    engage(arg(1))).
recommended_t([captain,jack],
    [aircraft,
     huffer,
     pilot],
    [running(parg(1),engine)],
    start(parg(1),engine)).
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [on(parg(1),'APU')],
    start(parg(1),'APU')).
recommended_t([captain,jack],
    [aircraft],
    [on(parg(1),'APU''s',generator)],
    engage(parg(1),'APU''s',generator)).
recommended_t([captain,jack],
    [aircraft],
    [on(parg(1),'APU''s',bleed,air)],
    engage(parg(1),'APU''s',bleed,air)).
recommended_t([captain,jack],
    [huffer,
     aircraft],
    [disengaged(arg(1))],
    disengage(arg(1))).
recommended_t([captain,jack],
    [aircraft,
     huffer,
     pilot],
    [off(parg(1),external,power)],
    disengage(parg(1),external,power)).
recommended_t([captain,jack],
    [aircraft,
     pilot,
     huffer],
    [checked(parg(1),'NWS')],
    check(parg(1),'NWS')).
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [checked(parg(1),brakes)],
    check(parg(1),brakes)).
recommended_t([captain,jack],
    [pilot,
     aircraft],
    ['cleared to depart'(arg(1))],
    request(flight,clearance)).
recommended_t([captain,jack],
    [pilot,
     aircraft],
    ['cleared for taxi'(arg(1))],
    request(taxi,clearance)).
```

```
recommended_t([captain,jack],
    [aircraft,
     pilot],
    ['on the runway'(arg(1))],
    taxi(arg(1))).
recommended_t([captain,jack],
    [pilot,
     aircraft],
    ['cleared for takeoff'(arg(1))],
    request(takeoff,clearance)).
recommended_t([captain,jack],
    [[wind,sock],
     pilot,
     aircraft],
    [checked(arg(1))],
    check(arg(1))).
recommended_t([captain,jack],
    [aircraft,
     pilot,
     [wind,sock]],
    [checked(parg(1),trim)],
    adjust(parg(1),trim)).
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [off(parg(1),'APU')],
    shut(off,parg(1),'APU')).
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [full(parg(1),throttles)],
    max(parg(1),throttles)).
recommended_t([captain,jack],
    [aircraft,
     pilot],
    [airborne(arg(1))],
    fly(the,arg(1))).
recommended_t([captain,jack],
    [aircraft],
    [off(parg(1),'APU''s',generator)],
    disengage(parg(1),'APU''s',generator)).
recommended_t([captain,jack],
    [aircraft],
    [off(parg(1),'APU''s',bleed,air)],
    disengage(parg(1),'APU''s',bleed,air)).
recommended_t([captain,jack],
    [aircraft],
    [off(parg(1),engine)],
    shut(off,parg(1),engine)).
recommended_t([captain,jack],
    [aircraft],
    [off(parg(1),throttles)],
    cut(parg(1),throttles)).
recommended_t([captain,jack],
    [aircraft],
    [checked(parg(1),trim)],
    check(parg(1),trim)).


/* precondition_t/5 */
precondition_t([captain,jack],
```

```
    [[wind,sock],
     pilot,
     aircraft],
    check(arg(1)),
    [],
    ['on the runway'(arg(3))]).
precondition_t([captain,jack],
    [huffer,
     aircraft],
    disengage(arg(1)),
    [],
    [on(parg(2),'APU''s',bleed,air),
     engaged(arg(1)),
     present(arg(1))]).
precondition_t([captain,jack],
    [huffer,
     aircraft],
    engage(arg(1)),
    [],
    [on(parg(2),external,power),
     disengaged(arg(1)),
     present(arg(1)),
     'preflight inspected'(arg(2))]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    taxi(arg(1)),
    [],
    ['cleared for taxi'(arg(2)),
     not('on the runway'(arg(1)))]).
precondition_t([captain,jack],
    [aircraft,
     pilot,
     [wind,sock]],
    adjust(parg(1),trim),
    [],
    [checked(arg(3)),
     unchecked(parg(1),trim)]).
precondition_t([captain,jack],
    [aircraft,
     pilot,
     huffer],
    check(parg(1),'NWS'),
    [],
    [off(parg(1),external,power),
     unchecked(parg(1),'NWS')]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    check(parg(1),brakes),
    [],
    [checked(parg(1),'NWS'),
     unchecked(parg(1),brakes)]).
precondition_t([captain,jack],
    [aircraft],
    check(parg(1),trim),
    [],
    [unchecked(parg(1),trim)]).
precondition_t([captain,jack],
    [aircraft],
    cut(parg(1),throttles),
```

```
    [],
    [not(off(parg(1),throttles))]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    fly(the,arg(1)),
    [],
    [full(parg(1),throttles),
     not(airborne(arg(1)))]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    max(parg(1),throttles),
    [],
    [off(parg(1),'APU'),
     not(full(parg(1),throttles)),
     'cleared for takeoff'(arg(2))]).
precondition_t([captain,jack],
    [pilot,
     aircraft],
    request(flight,clearance),
    [],
    [on(parg(2),'APU''s',bleed,air)]).
precondition_t([captain,jack],
    [pilot,
     aircraft],
    request(takeoff,clearance),
    [],
    ['on the runway'(arg(2)),
     'cleared for taxi'(arg(1))]).
precondition_t([captain,jack],
    [pilot,
     aircraft],
    request(taxi,clearance),
    [],
    ['cleared to depart'(arg(1)),
     checked(parg(2),brakes)]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    start(parg(1),'APU'),
    [],
    [running(parg(1),engine),
     off(parg(1),'APU'),
     'preflight inspected'(arg(1))]).
precondition_t([captain,jack],
    [aircraft,
     huffer,
     pilot],
    start(parg(1),engine),
    [],
    [engaged(arg(2)),
     off(parg(1),engine),
     'preflight inspected'(arg(1))]).
precondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',generator),
    [],
    [on(parg(1),'APU'),
     on(parg(1),'APU''s',generator)]).
precondition_t([captain,jack],
```

```
    [aircraft,
     buffer,
     pilot],
    disengage(parg(1),external,power),
    [],
    [disengaged(arg(2)),
     on(parg(1),external,power)]).
precondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',generator),
    [],
    [on(parg(1),'APU'),
     off(parg(1),'APU''s',generator)]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    engage(parg(1),external,power),
    [],
    ['preflight inspected'(arg(1)),
     off(parg(1),external,power)]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    shut(off,parg(1),'APU'),
    [],
    [checked(parg(1),trim),
     'cleared for takeoff'(arg(2)),
     on(parg(1),'APU')]).
precondition_t([captain,jack],
    [aircraft],
    shut(off,parg(1),engine),
    [],
    [running(parg(1),engine)]).
precondition_t([captain,jack],
    [aircraft,
     pilot],
    conduct(preflight,inspection,on,arg(1)),
    [],
    ['at the terminal'(arg(1))]).
precondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',bleed,air),
    [],
    [on(parg(1),'APU''s',generator),
     on(parg(1),'APU''s',bleed,air)]).
precondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',bleed,air),
    [],
    [on(parg(1),'APU''s',generator),
     off(parg(1),'APU''s',bleed,air)]).


/* deletepostcondition_t/5 */
deletepostcondition_t([captain,jack],
    [[wind,sock],
     pilot,
     aircraft],
    check(arg(1)),
    [],
    []).
```

```
deletepostcondition_t([captain,jack],
    [huffer,
     aircraft],
    disengage(arg(1)),
    [],
    [engaged(arg(1))]).
deletepostcondition_t([captain,jack],
    [huffer,
     aircraft],
    engage(arg(1)),
    [],
    [disengaged(arg(1))]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    taxi(arg(1)),
    [],
    ['at the terminal'(arg(1)),
     airborne(arg(1))]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot,
     [wind,sock]],
    adjust(parg(1),trim),
    [],
    [unchecked(parg(1),trim)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot,
     huffer],
    check(parg(1),'NWS'),
    [],
    [unchecked(parg(1),'NWS')]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    check(parg(1),brakes),
    [],
    [unchecked(parg(1),brakes)]).
deletepostcondition_t([captain,jack],
    [aircraft],
    check(parg(1),trim),
    [],
    [unchecked(parg(1),trim)]).
deletepostcondition_t([captain,jack],
    [aircraft],
    cut(parg(1),throttles),
    [],
    [idle(parg(1),throttles),
     full(parg(1),throttles)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    fly(the,arg(1)),
    [],
    ['at the terminal'(arg(1)),
     'on the runway'(arg(1))]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    max(parg(1),throttles),
```

```
    [],
    [off(parg(1),throttles),
     idle(parg(1),throttles)]).
deletepostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(flight,clearance),
    [],
    []).
deletepostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(takeoff,clearance),
    [],
    []).
deletepostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(taxi,clearance),
    [],
    []).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    start(parg(1),'APU'),
    [],
    [off(parg(1),'APU')]).
deletepostcondition_t([captain,jack],
    [aircraft,
     buffer,
     pilot],
    start(parg(1),engine),
    [],
    [off(parg(1),engine)]).
deletepostcondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',generator),
    [],
    [on(parg(1),'APU''s',generator)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     buffer,
     pilot],
    disengage(parg(1),external,power),
    [],
    [on(parg(1),external,power)]).
deletepostcondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',generator),
    [],
    [off(parg(1),'APU''s',generator)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    engage(parg(1),external,power),
    [],
    [off(parg(1),external,power)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    shut(off,parg(1),'APU'),
```

```
    [],
    [on(parg(1),'APU')]).
deletepostcondition_t([captain,jack],
    [aircraft],
    shut(off,parg(1),engine),
    [],
    [running(parg(1),engine)]).
deletepostcondition_t([captain,jack],
    [aircraft,
     pilot],
    conduct(preflight,inspection,on,arg(1)),
    [],
    []).
deletepostcondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',bleed,air),
    [],
    [on(parg(1),'APU''s',bleed,air)]).
deletepostcondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',bleed,air),
    [],
    [off(parg(1),'APU''s',bleed,air)]).


/* addpostcondition_t/5 */
addpostcondition_t([captain,jack],
    [[wind,sock],
     pilot,
     aircraft],
    check(arg(1)),
    [],
    [checked(arg(1))]).
addpostcondition_t([captain,jack],
    [huffer,
     aircraft],
    disengage(arg(1)),
    [],
    [disengaged(arg(1))]).
addpostcondition_t([captain,jack],
    [huffer,
     aircraft],
    engage(arg(1)),
    [],
    [engaged(arg(1))]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    taxi(arg(1)),
    [],
    ['on the runway'(arg(1))]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot,
     [wind,sock]],
    adjust(parg(1),trim),
    [],
    [checked(parg(1),trim)]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot,
```

```
     huffer],
    check(parg(1),'NWS'),
    [],
    [checked(parg(1),'NWS')]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    check(parg(1),brakes),
    [],
    [checked(parg(1),brakes)]).
addpostcondition_t([captain,jack],
    [aircraft],
    check(parg(1),trim),
    [],
    [checked(parg(1),trim)]).
addpostcondition_t([captain,jack],
    [aircraft],
    cut(parg(1),throttles),
    [],
    [off(parg(1),throttles)]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    fly(the,arg(1)),
    [],
    [airborne(arg(1))]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    max(parg(1),throttles),
    [],
    [full(parg(1),throttles)]).
addpostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(flight,clearance),
    [],
    ['cleared to depart'(arg(1))]).
addpostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(takeoff,clearance),
    [],
    ['cleared for takeoff'(arg(1))]).
addpostcondition_t([captain,jack],
    [pilot,
     aircraft],
    request(taxi,clearance),
    [],
    ['cleared for taxi'(arg(1))]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    start(parg(1),'APU'),
    [],
    [on(parg(1),'APU')]).
addpostcondition_t([captain,jack],
    [aircraft,
     huffer,
     pilot],
    start(parg(1),engine),
```

```
    [],
    [running(parg(1),engine)]).
addpostcondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',generator),
    [],
    [off(parg(1),'APU''s',generator)]).
addpostcondition_t([captain,jack],
    [aircraft,
     huffer,
     pilot],
    disengage(parg(1),external,power),
    [],
    [off(parg(1),external,power)]).
addpostcondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',generator),
    [],
    [on(parg(1),'APU''s',generator)]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    engage(parg(1),external,power),
    [],
    [on(parg(1),external,power)]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    shut(off,parg(1),'APU'),
    [],
    [off(parg(1),'APU')]).
addpostcondition_t([captain,jack],
    [aircraft],
    shut(off,parg(1),engine),
    [],
    [off(parg(1),engine)]).
addpostcondition_t([captain,jack],
    [aircraft,
     pilot],
    conduct(preflight,inspection,on,arg(1)),
    [],
    ['preflight inspected'(arg(1))]).
addpostcondition_t([captain,jack],
    [aircraft],
    disengage(parg(1),'APU''s',bleed,air),
    [],
    [off(parg(1),'APU''s',bleed,air)]).
addpostcondition_t([captain,jack],
    [aircraft],
    engage(parg(1),'APU''s',bleed,air),
    [],
    [on(parg(1),'APU''s',bleed,air)]).


/* randchange_t/7 */


/* singular_t/2 */


/* plural_t/2 */
```

```
/* apply_text_t/4 */
```

APPENDIX E. SCRIPT RUN OF METUTOR ON AN MEBUILDER FILE

This appendix shows METutor being run on the lesson produced in Appendix C and provided in Appendix D, called "pilot training I".

```
Script started on Wed Jul 27 14:11:36 1994
.alias: No such file or directory.
> ~galvint/mebuild/METutor
Name the lesson file> pilot_training_I
+-----------------------------------------------------------+
|  Means-Ends Tutoring System -- Version 29       (METutor)  |
+-----------------------------------------------------------+
|     by Professor Rowe and CPT Galvin,  Naval PG School     |
+-----------------------------------------------------------+

Welcome.  The name of this lesson is "pilot training I".
---------------------------------------
PILOT TRAINING:  LESSON 1

    This lesson is the first lesson in flying an aircraft. After this
lesson you will be familiar with the process of starting the plane and
taking off.  The specific skills taught in this lesson are:

        (a)  Conducting all preflight checks and inspections

        (b)  Basic communications with the tower

    There is one problem in the lesson, a comprehensive test of the
skill.  Good luck.


---------------------------------------
There are 3 problems in the lesson.
You may "list" the problems, "view" a summary of a problem,
or "do" a problem.  "help" is also available.
METutor> help
-----------------------------------------------------------
The following commands are available at this level:
   help            -- print this text
   quit            -- quit METutor
   list            -- lists all the problems available.
   learn           -- <not yet implemented -- this is a hook
                       for future use in administering lessons.>
```

```
   view [problem <number>]
                    -- prints a description of the problem.
   do [problem <number>]
                    -- run a problem.
If the optional argument is not supplied, METutor will ask
for it.  A different command set is available while running
a problem.
------------------------------------------------------------
METutor> view problem 3
The name of the problem is "Comprehensive Test".

Description:
Now that you have successfully completed the various phases of the
process, let's put the whole thing together from the start.  Good
luck!


METutor> do problem 3
Loading and checking the problem.....please wait....Done.
The following is the current situation:
---------------------------------------
Now that you have successfully completed the various phases of the
process, let's put the whole thing together from the start.  Good
luck!


The following are your objectives:
you must be cleared to depart,  you must be cleared for taxi, you must
be
      cleared for takeoff,  B-10 must be preflight inspected,  the
      aircraft must be airborne,  B-10's brakes must be checked,  the
      aircraft's engine must be running,  B-10's NWS must be
      checked,  B-10's throttles must be full,  B-10's trim
      must be checked,  the huffer must be present,  and the wind sock
      must be checked

The following is the current situation:
B-10 is at the terminal,  the huffer is disengaged,  the huffer is
      present,  B-10's APU is off,  B-10's engine is off,
      B-10's throttles are off,  B-10's NWS is unchecked,
      B-10's brakes are unchecked,  B-10's trim is
      unchecked,  B-10's external power is available, B-10's
      APU's generator is off,  B-10's external power is off,   and
      B-10's APU's bleed air is off

What do you want to do? help
------------------------------------------------------------
You may enter an operator or one of these special commands:
   help             -- print this text
   quit             -- return to METutor main level
   view state       -- pretty prints the current state
```

```
   view objectives -- pretty prints your objectives
   query operator <operator>
      -- prints all information about an operator.
   query object   <object>
      -- prints operators available on an object.
   query fact     <object>
      -- prints all information about a fact or objective.
The following are the operators available in this lesson:
[1]   adjust B-10's trim
[2]   check B-10's NWS
[3]   check B-10's brakes
[4]   check the wind sock
[5]   conduct preflight inspection on B-10
[6]   disengage B-10's external power
[7]   disengage the huffer
[8]   engage B-10's APU's bleed air
[9]   engage B-10's APU's generator
[10]  engage B-10's external power
[11]  engage the huffer
[12]  fly the B-10
[13]  max B-10's throttles
[14]  request flight clearance
[15]  request takeoff clearance
[16]  request taxi clearance
[17]  shut off B-10's APU
[18]  start B-10's APU
[19]  start B-10's engine
[20]  taxi B-10
------------------------------------------------------------
What do you want to do? view operator start B-10's APU
Sorry, that is not a valid command.  Please try again.
What do you want to do? query operator start B-10's APU
The following is true about "start B-10's APU":
** The operator is recommended for achieving "B-10's APU is on "
** The precondition for the operator is "B-10's engine must be
      running,  B-10's APU must be off,   and B-10 must be
      preflight inspected ".
** The postcondition for the operator is "B-10's APU is no longer off
" while "B-10's APU would be on ".
What do you want to do? query fact B-10's external power is on

The following operators are recommended for achieving this fact:
[1]   engage B-10's external power

What do you want to do? query object B-10

The following can be performed on "B-10".
** Operator = "conduct preflight inspection on B-10":
   The operator is intended to achieve "B-10 would be
      preflight inspected ".
```

```
** Operator = "taxi B-10":
   The operator is intended to achieve "B-10 would be on the runway ".

** Operator = "fly the B-10":
   The operator is intended to achieve "B-10 would be airborne ".

What do you want to do? conduct preflight inspection on B-10
You chose to conduct preflight inspection on B-10.

OK.
[1]   B-10 is now preflight inspected

What do you want to do? start B-10's APU
You chose to start B-10's APU.

Have you confused "B-10's engine is running " with "B-10's
      engine is off "?
That action requires that:
[1]   B-10's engine must be running

What do you want to do? start B-10's engine
You chose to start B-10's engine.

Have you confused "the huffer is engaged " with "the huffer is
disengaged "?
That action requires that:
[1]   the huffer must be engaged

What do you want to do? engage the huffer
You chose to engage the huffer.

Have you confused "B-10's external power is on " with "B-10's
      external power is available "?
That action requires that:
[1]   B-10's external power must be on

What do you want to do? engage B-10's external power
You chose to engage B-10's external power.

OK.
[1]   B-10's external power is now on

What do you want to do? engage the huffer
You chose to engage the huffer.

OK.
[1]   the huffer is now engaged

What do you want to do? start B-10's engine
You chose to start B-10's engine.
```

```
OK.
[1]   B-10's engine is now running

What do you want to do? request flight clearance
You chose to request flight clearance.

Have you confused "B-10's APU's bleed air is on " with "B-10's APU's
bleed air is off "?
That action requires that:
[1]   B-10's APU's bleed air must be on

What do you want to do? query fact B-10's APU's bleed air is on

The following operators are recommended for achieving this fact:
[1]   engage B-10's APU's bleed air

What do you want to do? engage B-10's APU's bleed air
You chose to engage B-10's APU's bleed air.

Have you confused "B-10's APU's generator is on " with "B-10's APU's
generator is off "?
That action requires that:
[1]   B-10's APU's generator must be on

What do you want to do? quit
METutor> quit
> exit
> exit

script done on Wed Jul 27 14:15:26 1994
```

APPENDIX F. EXPERIMENT CONDUCTED USING MEBUILDER

This Appendix contains the raw information produced and gathered in the process of conducting the validity experiments during the Summer Quarter of 1994. This experiment is discussed in thorough detail in Chapter V. The following lists the information in this Appendix.

Tab 1. General Instructions for the Experiment

Tab 2. Suite One: The Scuba Diving Problem

Tab 3. Suite Two: The Cooling System Problem

Tab 4. Sample Run of the Data Collection Program

Tab 5. Initial Data Files for the Scuba Diving Problem

Tab 6. Initial Data Files for the Cooling System Problem

Tab 7. Data Collected

Tab 8. Selected Comments from Participants

TAB 1. GENERAL INSTRUCTIONS FOR THE EXPERIMENT

ASSIGNMENT

You will be required to construct a lesson in a simple procedural task using two different tools. One tool is based on the principles of Computer-Aided Instruction (CAI), the other using a intelligent lesson authoring system (MEBuilder).

PURPOSE OF THE EXPERIMENT

The purpose of the experiment is to gather evidence concerning how well MEBuilder helps teachers write lessons versus older methods. This evidence will be interpreted based on raw data produced from the following:

a. Amount of time required to produce the lesson material in each platform. This will be measured in terms of raw time and number of steps required.

b. The completeness of each lesson -- whether or not MEBuilder hindered the writing process to the point that the desired lesson could not be written satisfactorily.

c. The robustness of each lesson -- whether or not the resulting lesson affords the student the maximum or correct numbers of choices at any point while running the lesson.

CONDUCT OF THE EXPERIMENT

The experiment will be conducted as follows:

a. Orientation. This document will be presented and classroom instruction given on the differences between CAI and ICAI techniques. This will be followed by a detailed block of instruction on the use of two tools -- CAIBuilder and MEBuilder -- introduced later in the text.

b. Conduct. A set amount of time will be allowed for students to build the lesson material and to gather the necessary data as requested below. Handed out separately is the specific subject matter the student will be required to author a lesson on.

You will be given a library with half-completed solutions in it. The files include a CAI-solution where there is exactly one path to the goal and no options given to the student, and an MEBuilder-solution where there is exactly one path to the goal and no options given to the student. Your job is to make both lessons robust in order to conform more closely to the task descriptions given.

NOTE: This experiment is intended to require no more than six hours of application running time. This time includes familiarization with the two systems. If you are having serious problems performing the requirements under six hours, contact galvint by email as soon as possible for assistance.

c. Debriefing. A forum will be held for students to provide specific general comments about the experiment. In addition, the experimentor will provide additional data relating the students' experiences with the expected or optimistic results.

TOOLS AND DOCUMENTATION PROVIDED

a. General Information. Each student will have a copy of this document plus a copy of the specific subject matter for his lessons. You must establish a single subdirectory for this project, and you must run all the below listed programs from within this subdirectory. You are free to copy the executables into your own directory (it totals to about 3.6MB).

b. CAITutor and CAIBuilder tools. Each student will have a copy of the user's manual for the CAITutor and CAIBuilder systems. The programs are available in executable form in **~galvint/caitutor** and are called **CAITutor** and **CAIBuilder**, respectively.

c. METutor and MEBuilder. Likewise, the student's will have a copy of the user's manual for METutor and MEBuilder systems. The programs are available in executable form in **~galvint/mebuild** and are called **METutor** and **MEBuilder**, respectively.

d. Statistical Gathering Programs. The student's will have access to **get_data** which is a simple five-step program that retrieves statistical information from the student's directory, queries some time information of the student, and then prints a standard data report for analysis. **get_data** is located in the **~galvint/mebuild** directory.

e. User's Manuals. User's Manuals for CAIBuilder and MEBuilder will be available in binders in the AI lab (they may be available individually). The user's manuals contain a description of the programs, complete command references, and sample sessions using a lesson with a scope similar to that of the assigned lesson.

f. Library. In the **~galvint/sample/lib/** directory is all of the preliminary data you will need. It contains the skeletal lessons for both the CAI solution (*.cai) and the MEBuilder solution. You are to do the following:

(1) Copy the directory into an lib subdirectory. It must be named lib, so if you are ~student and you intend to work in the **~student/cs4310** directory, then you must put this library in the **~student/cs4310/lib** subdirectory and you must run MEBuilder from **~student/cs4310**.

(2) Move the appropriate .cai file into the library's parent directory (in the above example, it would be **~student/cs4310**.

Note: Included in the library are the sample lessons built under the demonstration portions of the two manuals (*prep_aircraft.cai* and *pilot_training_1.les*, respectively) and *pilot_training_1.met* is also available for running in **METutor**).

DELIVERABLES

The required deliverables are a summary of your work with the two systems including comments about the interface, brief script runs of the lessons being run in CAITutor and METutor (no need to show a complete run, just enough to show some of the changes you made) and the output of the get_data program. The summary should not exceed two pages in length.

NOTES CONCERNING THE USE OF MEBUILDER

In order to ensure that the statistical measurements are accurate, your use of MEBuilder must conform to the following rules:

a. The MEBuilder lesson will contain precisely one problem.

b. The MEBuilder lesson will be based on precisely one task, which encompasses the entire procedure being taught.

You are free to experiment with the MEBuilder lesson structure once the deliverable statistical information has been gathered.

TAB 2. SUBJECT MATTER FOR THE EXPERIMENT -- SUITE ONE

Lesson One. The Scuba-Diving Problem

The student is a scuba diver who plans to dive for lobster from an anchored boat. The lesson focuses on the ability of the student to self-equip and descend to the sea floor to get the lobster.

At the start of the problem, the student will be inside a boat wearing no scuba gear. At the end of the problem, the student should be located on the sea floor with the lobster in his possession.

Description of the Procedure

For those of you who actually scuba dive, you will note that this is a subset of the actual procedure used and the ordering of steps is more restrictive than in a real situation. The scope has been reduced in order to make the size of the problem manageable.

The student has the following gear present -- a knife, an air regulator, an air tank, a weightbelt, fins, and a mask. Also present is the diver's buddy. You do not specifically have to model the buddy, boat, or the lobster (ways to do this are given below). The task is as follows:

(1) The diver must in order install then test the regulator, and mounts it on the air tank.

(2) The diver dons the knife and the weightbelt and dons the air tank. These three stepe can be done in any order.

(3) The student then dons the fins and mask, and checks his buddy's tank. These three steps can be done in any order.

(4) The student then in order valsalvos to clear his sinuses, enters the water, sets the tank to negative buoyancy by releasing some air, then descends to the sea floor and bags the lobster.

Hints and Helpful Information

For MEBuilder, provided are the objects "diver", "knife", "air regulator", "air tank", "weightbelt", "fins", and "mask"; along with the task "prepare diver". The lesson is not provided, you will build it. The CAI lesson is in prepare_diver.cai.

In CAIBuilder, the states in the lesson are numbered in order start, 1, ... , done. When following Appendix B of the CAIBuilder manual, pages 4 and most of 5 are already done. Your requirements begin at the bottom of page 5.

In MEBuilder, you are starting at the point where the initial solution is completed, bottom of page 54. Do the work on task named prepare diver command to reach this point.

TAB 3. SUBJECT MATTER FOR THE EXPERIMENT -- SUITE TWO

Lesson Two. The Cooling System Problem

You are presenting the student with a car with a leaky gasket in the water pump. His job is to complete the task of removing the water pump, replacing the gasket, and restoring the car to service.

The start state is that all parts of the engine are in their normal configuration -- that is to say the radiator is filled with fluid, the hoses are attached, the belts are in place, etc., etc. You want the student to have restored the engine's condition with the exception of the new gasket being in place.

Description of the Procedure

For those of you who actually work on cars, you will note that this is a subset of the actual procedure used and the ordering of steps is more restrictive than in a real situation. The scope has been reduced in order to make the size of the problem manageable.

The student is only going to be concerned with the following items on the car -- the engine fan, the radiator, the radiator's hoses, the belts, and the water pump. You do not specifically have to model the car. The task is as follows:

(1) The mechanic must do the following two tasks in either order:
  (a) Unbolt, then remove the fan
  (b) Drain the radiator, then remove the hoses

(2) The mechanic must then do the following in sequence -- remove the belts, unbolt the pump, remove it, replace the gasket, install the pump, then bolt it in place, andreinstall the belts.

(3) The mechanic must then do the inverse of Step (1) above. In either order:
  (a) Install the fan, then bolt it in place
  (b) Install the hoses, then fill the radiator

Hints and Helpful Information

For MEBuilder, provided are the objects "fan", "radiator", "hoses", "water pump"; along with the task "replace gasket". The lesson is not provided, you will build it. The CAI lesson is in replace_gasket.cai.

In CAIBuilder, the states in the lesson are numbered in order start, 1, ... , done. When following Appendix B of the CAIBuilder manual, pages 4 and most of 5 are already done. Your requirements begin at the bottom of page 5.

In MEBuilder, you are starting at the point where the initial solution is completed, bottom of page 54. Do the work on task named replace gasket command to reach this point.

TAB 4. SAMPLE RUN OF THE DATA COLLECTION PROGRAM

In order to use the **get_data** program, you must be in the directory that contains the special file **user.cfg**. The task created from **MEBuilder** must be in the "lib" subdirectory. The resulting text file will appear in the **mebuild.rpt** file in the working directory. The following is a **get_data** session with user inputs highlighted.

```
gemini:/users/work1/galvint/sample>> ~galvint/mebuild/get_data
+-----------------------------------------------------------+
| MEBUILDER Experiment Data Retrieval and Interpretation |
|              Program -- Experiment of SQ 94              |
+-----------------------------------------------------------+
Name the CAI file> flashlight_repair.cai
I found 2 solutions to the task.
The solution has 14 nodes and 16 transitions.
Give the TASK NAME -- not the file name
Name the MEBuilder task> flashlight repair
I found 2 solutions to the task.
The solution has 8 nodes and 7 transitions.
You will now be asked a series of questions regarding the amount of
time you spent using MEBuilder and CAIBuilder and how that time was
spent.  When you entire "quit", a report file named "mebuild.rpt"
will be produced.  This report should be submitted with the rest
of the experimental deliverables.
DO NOT include time lost due to CAIBuilder or MEBuilder program bugs.

Please give integer values for the following.  Include time spent
reading through the materials, practicing, editing, testing, etc.
How many hours did you spend on the CAI task? 12
How many hours did you spend on the MEB task? 10

You will now provide a rating list of the areas within the building
process that you spent time on.
Please express your answer as a *permutation* of the following letters:
        f       d       e       t
...the order must be of most time spent to least time spent
f=familiarization.  Reading the user's manual, and running practice
    sessions in an attempt to get accustomed to the process.
d=designing.  Time spent designing the objects on paper.
e=entering.   Time spent entering the lesson data
t=testing.  Time spent testing and "debugging" the end result.

Again, please answer as a permutation of [f,d,e,t] in order from
most time spent to least time spent.
How did you spend your time using CAIBuilder> e d t f
How did you spend your time using MEBuilder > t e d f

For MEBuilder, provide your response the same way, but only for the
named subportion of the process:
Building objects in MEBuilder   > e f t d
Building the task in MEBuilder  > t d e f
```

```
Building the lesson in MEBuilder> t e d f
The report has been printed.  Thank you for your participation.
```

The following is the sample mebuild.rpt produced from the session.

```
gemini:/users/work1/galvint/sample>> more mebuild.rpt
                    MEBuilder Experiment Report
                         Summer Quarter 94


1.  Command Usage Comparison
                                        CAI      MEB
      Number of Commands Performed:     84       95
      Number of Commands Aborted:       6        11
      Percentage of Aborted Commands:   7.14%    11.57%

2.  Time Usage and Effectiveness
                                        CAI          MEB
      Total time (in hours):            12           10

      Most Time-Consuming Process:      entering     testing
      2nd Most Time-Consuming Process:  designing    entering
      3rd Most Time-Consuming Process:  testing      designing
      Least Time-Consuming Process:     familiar     familiar

      For the components of the MEBuilder process:
                                  OBJECTS     TASKS       LESSONS
      Most Time-Consuming:        entering    testing     testing
      2nd Most Time-Consuming:    familiar    designing   entering
      3rd Most Time-Consuming:    testing     entering    designing
      Least Time-Consuming:       designing   familiar    familiar

3.  Lesson Material Produced
                                        CAI      MEB
     Number of Solutions in Lesson:     2        2
     Number of Nodes:                   14       8
     Number of Transitions:             16       7
gemini:/users/work1/galvint/sample>>
```

TAB 5. INITIAL DATA FILES FOR SUITE ONE

This tab contains the CAIBuilder lesson and the MEBuilder task given to the students for suite one of the experiment. The MEBuilder object files are not included since the students will not modify them as part of the experiment.

The CAIBuilder files consist of ***cai_node(<node>)*** facts which declare the valid nodes, and ***cai_step(<source node>, <operation>, <message>, <destination node>)*** which declare the valid transitions. As with MEBuilder, the *start* and *done* nodes are reserved for the beginning and completion of the task. The convention used for CAIBuilder is to name the nodes the same as the MEBuilder steps in the task.

ORIGINAL CAIBUILDER SOLUTION

```
:- dynamic      type_of_prolog_file/1.

:- multifile    type_of_prolog_file/1.

:- dynamic      cai_intro/1, cai_step/4, cai_node/1.

:- multifile    cai_intro/1, cai_step/4, cai_node/1.


type_of_prolog_file('CAIBuild Lesson Definition File').


/* cai_intro/1 */
cai_intro('    You are a scuba diver who plans to dive for lobster from an').
cai_intro('anchored boat.  This lesson will focus on your ability to self-').
cai_intro('equip and descend to the save floor to get the lobster.').
cai_intro('').
cai_intro('    At the beginning, you will be in the boat wearing no scuba').
cai_intro('gear.  The gear present includes a knife, a regulator, an air').
cai_intro('tank, a weightbelt, fins, and a mask.  You also have a diving').
cai_intro('buddy present.').
cai_intro('').
cai_intro('    The lesson will end once you have reached the sea floor and').
cai_intro('bagged the lobster.  Good luck!').


/* cai_step/4 */
cai_step(start,[install,the,regulator],
    ['The',regulator,is,now,installed],
    2).
cai_step(2,[turn,on,the,air],
    ['The',air,is,now,turned,on],
    3).
cai_step(3,[test,the,regulator],
    ['The',regulator,is,working,fine],
    4).
cai_step(4,[don,the,knife],
    ['You',are,now,wearing,the,knife],
    5).
cai_step(5,[don,the,weightbelt],
    ['You',are,now,wearing,the,weightbelt],
    6).
cai_step(6,[don,the,air,tank],
```

```
    ['You',are,now,wearing,the,air,tank],
    7).
cai_step(7,[don,the,fins],
    ['You',are,now,wearing,the,fins],
    8).
cai_step(8,[don,the,mask],
    ['You',are,now,wearing,the,mask],
    9).
cai_step(9,[check,your,'buddy''s',tank],
    ['Your','buddy''s',tank,appears,to,be,'OK'],
    10).
cai_step(11,[release,air],
    ['Your',tank,is,now,negatively,buoyant],
    12).
cai_step(10,[valsalvo],
    ['You',have,cleared,your,sinuses],
    '10A').
cai_step('10A',[enter,the,water],
    ['You',are,now,in,the,water],
    11).
cai_step(12,[descend],
    ['You',are,now,on,the,sea,floor],
    13).
cai_step(13,[bag,the,lobster],
    ['You',have,captured,a,nice,big,juicy,lobster],
    done).


/* cai_node/1 */
cai_node(start).
cai_node(done).
cai_node(2).
cai_node(3).
cai_node(4).
cai_node(5).
cai_node(6).
cai_node(7).
cai_node(8).
cai_node(9).
cai_node(10).
cai_node(11).
cai_node(12).
cai_node('10A').
cai_node(13).
```

ORIGINAL MEBUILDER TASK

```
/*****************************************************************/
/*                MEBuilder Task Definition File                 */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.

:- dynamic     task/3, initial_conditions/2, objectives/2, stage/6,
```

```
                action/6, unordered_action/2, relation/3.

:- multifile    task/3, initial_conditions/2, objectives/2, stage/6,
                action/6, unordered_action/2, relation/3.


type_of_prolog_file('MEBuilder Library Task Definition File').


/* task/3 */
task([prepare,diver],[diver,diver],
    [[lobster,lobster]]).


/* initial_conditions/2 */
initial_conditions(diver,
    [not('buddy checked'(diver)),
     not(cleared(diver)),
     'in the boat'(diver),
     doffed('diver''s',weightbelt),
     doffed('diver''s',knife),
     doffed('diver''s',mask),
     doffed('diver''s',fins),
     removed('diver''s',regulator),
     not(tested('diver''s',regulator)),
     off('diver''s',air,tank),
     doffed('diver''s',air,tank),
     'positively buoyant'('diver''s',air,tank)]).
initial_conditions(lobster,
    [free(lobster)]).


/* objectives/2 */
objectives(diver,
    ['buddy checked'(diver),
     cleared(diver),
     'on the sea floor'(diver),
     donned('diver''s',weightbelt),
     donned('diver''s',knife),
     donned('diver''s',mask),
     donned('diver''s',fins),
     installed('diver''s',regulator),
     tested('diver''s',regulator),
     on('diver''s',air,tank),
     donned('diver''s',air,tank),
     'negatively buoyant'('diver''s',air,tank)]).
objectives(lobster,
    [captured(lobster)]).


/* stage/6 */
stage(start,linear,none,linear,
    [],
    []).
stage(q1,linear,none,linear,
    [[start,101,1]],
    []).
stage(q2,linear,none,linear,
    [[q1,102,1]],
    []).
stage(q3,linear,none,linear,
```

```
    [[q2,103,1]],
    []).
stage(q4,linear,none,linear,
    [[q3,104,1]],
    []).
stage(q5,linear,none,linear,
    [[q4,105,1]],
    []).
stage(q6,linear,none,linear,
    [[q5,106,1]],
    []).
stage(q7,linear,none,linear,
    [[q6,107,1]],
    []).
stage(q8,linear,none,linear,
    [[q7,108,1]],
    []).
stage(q9,linear,none,linear,
    [[q8,109,1]],
    []).
stage(q10,linear,none,linear,
    [[q9,110,1]],
    []).
stage(q11,linear,none,linear,
    [[q10,111,1]],
    []).
stage(q12,linear,none,linear,
    [[q11,112,1]],
    []).
stage(q13,linear,none,linear,
    [[q12,113,1]],
    []).
stage(done,no-actions,none,linear,
    [[q13,114,1]],
    []).


/* action/6 */
action(101,install('diver''s',regulator),
    [],
    [],
    [],
    []).
action(102,test('diver''s',regulator),
    [],
    [],
    [],
    []).
action(103,turn(on,'diver''s',air,tank),
    [],
    [],
    [],
    []).
action(104,don('diver''s',weightbelt),
    [],
    [],
    [],
    []).
action(105,don('diver''s',knife),
    [],
    [],
```

```
    [],
    []).
action(106,don('diver''s',air,tank),
    [],
    [],
    [],
    []).
action(107,don('diver''s',fins),
    [],
    [],
    [],
    []).
action(108,don('diver''s',mask),
    [],
    [],
    [],
    []).
action(109,have(diver,check,buddy),
    [],
    [],
    [],
    []).
action(110,have(diver,valsalvo),
    [],
    [],
    [],
    []).
action(111,have(diver,enter,the,water),
    [],
    [],
    [],
    []).
action(112,release(air,from,'diver''s',air,tank),
    [],
    [],
    [],
    []).
action(113,have(diver,descend),
    [],
    [],
    [],
    []).
action(114,bag(lobster),
    [],
    [],
    [],
    []).


/* unordered_action/2 */


/* relation/3 */
```

TAB 6. INITIAL DATA FILES FOR SUITE TWO

This tab contains the CAIBuilder lesson and the MEBuilder task given to the students for suite two of the experiment. The MEBuilder object files are not included since the students will not modify them as part of the experiment.

The CAIBuilder files consist of *cai_node(<node>)* facts which declare the valid nodes, and *cai_step(<source node>, <operation>, <message>, <destination node>)* which declare the valid transitions. As with MEBuilder, the *start* and *done* nodes are reserved for the beginning and completion of the task. The convention used for CAIBuilder is to name the nodes the same as the MEBuilder steps in the task.

ORIGINAL CAIBUILDER LESSON

```
:- dynamic      type_of_prolog_file/1.

:- multifile    type_of_prolog_file/1.

:- dynamic      cai_intro/1, cai_step/4, cai_node/1.

:- multifile    cai_intro/1, cai_step/4, cai_node/1.


type_of_prolog_file('CAIBuild Lesson Definition File').


/* cai_intro/1 */
cai_intro('    Before you is a car with a leaky gasket in the water pump.').
cai_intro('Your job is to replace the gasket and restore the car to its').
cai_intro('working condition.').
cai_intro('').
cai_intro('    The car presently has its fan, belts, and hoses installed.').
cai_intro('The radiator is full, and the water pump is in place.').
cai_intro('').
cai_intro('    Good luck.').


/* cai_step/4 */
cai_step(start,[unbolt,the,fan],
    ['The',fan,is,now,free],
    2).
cai_step(2,[remove,the,fan],
    ['The',fan,is,now,removed,from,the,car],
    3).
cai_step(3,[drain,the,radiator],
    ['The',radiator,is,now,free,of,liquid],
    4).
cai_step(4,[remove,the,hoses],
    ['The',hoses,have,been,removed,from,the,radiator],
    5).
cai_step(5,[remove,the,belts],
    ['The',belts,have,been,removed],
    6).
cai_step(6,[unbolt,the,water,pump],
    ['The',water,pump,is,now,free],
```

```
    7).
cai_step(7,[remove,the,water,pump],
    ['The',water,pump,is,now,removed,from,the,car],
    8).
cai_step(8,[replace,the,gasket],
    ['The',new,gasket,has,now,been,installed],
    9).
cai_step(9,[install,the,water,pump],
    ['The',water,pump,is,now,reinstalled],
    10).
cai_step(10,[bolt,the,water,pump],
    ['The',water,pump,is,now,secured],
    11).
cai_step(11,[install,the,belts],
    ['The',belts,are,now,installed],
    12).
cai_step(12,[install,the,fan],
    ['The',fan,is,now,in,place],
    13).
cai_step(13,[bolt,the,fan],
    ['The',fan,is,now,bolted,in,place],
    14).
cai_step(14,[install,the,hoses],
    ['The',hoses,are,now,installed],
    15).
cai_step(15,[fill,the,radiator],
    ['The',radiator,is,now,'filled.'],
    done).


/* cai_node/1 */
cai_node(start).
cai_node(done).
cai_node(2).
cai_node(3).
cai_node(4).
cai_node(5).
cai_node(6).
cai_node(7).
cai_node(8).
cai_node(9).
cai_node(10).
cai_node(11).
cai_node(12).
cai_node(13).
cai_node(14).
cai_node(15).
```

ORIGINAL MEBUILDER TASK

```
/*****************************************************************/
/*                MEBuilder Task Definition File                 */
/*****************************************************************/


:- dynamic     type_of_prolog_file/1.

:- multifile   type_of_prolog_file/1.
```

```
:- dynamic     task/3, initial_conditions/2, objectives/2, stage/6,
               action/6, unordered_action/2, relation/3.

:- multifile   task/3, initial_conditions/2, objectives/2, stage/6,
               action/6, unordered_action/2, relation/3.


type_of_prolog_file('MEBuilder Library Task Definition File').


/* task/3 */
task([replace,gasket],[character,character],
    [[[car,engine],[car,engine]]]).


/* initial_conditions/2 */
initial_conditions(character,
    []).
initial_conditions([car,engine],
    [installed(car,'engine''s',belts),
     installed(car,'engine''s',hoses),
     worn(car,'engine''s',gasket),
     installed(car,'engine''s',water,pump),
     bolted(car,'engine''s',water,pump),
     filled(car,'engine''s',radiator),
     installed(car,'engine''s',fan),
     bolted(car,'engine''s',fan)]).


/* objectives/2 */
objectives(character,
    []).
objectives([car,engine],
    [installed(car,'engine''s',belts),
     installed(car,'engine''s',hoses),
     serviceable(car,'engine''s',gasket),
     installed(car,'engine''s',water,pump),
     bolted(car,'engine''s',water,pump),
     filled(car,'engine''s',radiator),
     installed(car,'engine''s',fan),
     bolted(car,'engine''s',fan)]).


/* stage/6 */
stage(start,linear,none,linear,
    [],
    []).
stage(q1,linear,none,linear,
    [[start,101,1]],
    []).
stage(q2,linear,none,linear,
    [[q1,102,1]],
    []).
stage(q3,linear,none,linear,
    [[q2,103,1]],
    []).
stage(q4,linear,none,linear,
    [[q3,104,1]],
    []).
stage(q5,linear,none,linear,
    [[q4,105,1]],
```

```
    []).
stage(q6,linear,none,linear,
    [[q5,106,1]],
    []).
stage(q7,linear,none,linear,
    [[q6,107,1]],
    []).
stage(q8,linear,none,linear,
    [[q7,108,1]],
    []).
stage(q9,linear,none,linear,
    [[q8,109,1]],
    []).
stage(q10,linear,none,linear,
    [[q9,110,1]],
    []).
stage(q11,linear,none,linear,
    [[q10,111,1]],
    []).
stage(q12,linear,none,linear,
    [[q11,112,1]],
    []).
stage(q13,linear,none,linear,
    [[q12,113,1]],
    []).
stage(q14,linear,none,linear,
    [[q13,114,1]],
    []).
stage(done,no-actions,none,linear,
    [[q14,115,1]],
    []).


/* action/6 */
action(101,unbolt(car,'engine''s',fan),
    [],
    [],
    [],
    []).
action(102,remove(car,'engine''s',fan),
    [],
    [],
    [],
    []).
action(103,drain(car,'engine''s',radiator),
    [],
    [],
    [],
    []).
action(104,remove(car,'engine''s',hoses),
    [],
    [],
    [],
    []).
action(105,remove(car,'engine''s',belts),
    [],
    [],
    [],
    []).
action(106,unbolt(car,'engine''s',water,pump),
    [],
```

```
    [],
    [],
    []).
action(107,remove(car,'engine''s',water,pump),
    [],
    [],
    [],
    []).
action(108,replace(car,'engine''s',gasket),
    [],
    [],
    [],
    []).
action(109,install(car,'engine''s',water,pump),
    [],
    [],
    [],
    []).
action(110,bolt(car,'engine''s',water,pump),
    [],
    [],
    [],
    []).
action(111,install(car,'engine''s',belts),
    [],
    [],
    [],
    []).
action(112,install(car,'engine''s',fan),
    [],
    [],
    [],
    []).
action(113,bolt(car,'engine''s',fan),
    [],
    [],
    [],
    []).
action(114,install(car,'engine''s',hoses),
    [],
    [],
    [],
    []).
action(115,fill(car,'engine''s',radiator),
    [],
    [],
    [],
    []).


/* unordered_action/2 */


/* relation/3 */
```

TAB 7. RAW EXPERIMENTAL DATA COLLECTED

For the row headers marked in *italics*, the following is the legend:

- D = Time spent designing the task (prior to running the authoring system)
- E = Time spent entering the necessary commands into the authoring system.
- F = Time spent familiarizing (reading the manuals and running the sample cases).
- T = Time spent testing and debugging the lesson in the ITS.

Table 1: Raw Data Collected from the Data Collection Program

Raw Data Per Subject	1	2	3	4	5	6
(D)iver Problem or (E)ngine Problem	E	D	D	E	D	D
(C)AIBuilder first or (M)EBuilder first	C	C	C	M	M	C
Time Spent Using CAIBuilder (hours)	2	3	3	2	1	3
Time Spent Using MEBuilder	3	2	2	1	1	2
Most Time-Consuming Process (CAI/MEB)	E/E	D/E	F/F	F/F	E/F	F/F
2nd Most Time-Consuming Process (CAI/MEB)	D/F	E/D	T/T	E/E	F/E	E/E
3rd Most Time-Consuming Process (CAI/MEB)	F/D	F/F	E/E	D/D	D/D	D/D
4th Most Time-Consuming Process (CAI/MEB)	T/T	T/T	D/D	T/T	T/T	T/T
Number of CAI Commands Performed	78	140	164	151	117	68
Number of CAI Commands Aborted	1	9	3	2	0	0
Percentage of Aborted Commands	1.3	6.42	1.82	1.33	0.0	0.0
Number of MEB Commands Performed	29	25	20	11	45	unk*
Number of MEB Commands Aborted	9	8	1	0	6	unk*
Percentage of Aborted Commands	31.03	32.0	5.0	0.0	13.3	unk*
Number of Solutions (actual = 36)	32	36	36	4	36	36
Number Nodes/Transitions in CAI Solution	26/33	23/32	23/32	23/38	23/32	23/32
Number Nodes/Transitions in MEB Solution	16/15	14/13	14/13	16/15	14/13	14/13

* - The data collection program failed for this participant's MEBuilder usage. The participant stated that his usage was somewhat on par with his peers.

The minimal number of CAI nodes and transitions were: for the diver problem, 23 nodes and 32 transitions; for the engine problem, 24 nodes and 31 transitions. All participants achieved the minimal MEBuilder data structure.

TAB 8. SELECTED COMMENTS FROM THE PARTICIPANTS

The vast majority of the comments were interface-related, mostly having to do with specific program glitches, the cumbersomeness of the command-line interface, or complaints about the help system (which was not fully updated in time for the experiment). A common theme among the interface comments was the call for a graphic-user interface.

The comments selected below were those which specifically addressed the focus of the experiment -- the respective learning curves and flexibility of the two methods.

From Participant #3:

"CAI: Initial I found the concept, manual, and help very confusing, but once I broke the code it went smoothly. MEB: [MEBuilder] was just as confusing if not more so than CAI. I was not sure what was being provided and what I needed to create. Once I broke the code though, it saved a lot of time as compared to CAI."

From Participant #4

"...Both are frustrating to learn (especially when you aren't too motivated). But once you get going, neither are too bad...Once I understood what to do, MEB was quick; however, I wonder if it would have been as easy if the post & pre conditions, etc. weren't already done...A menu-based application would be easier for the average computer-phobic to use."

From Participant #6

"...My general comments are that both systems seem fairly straight forward to use. Actually, MEBuilder seemed much more complicated and I don't know the system well enough to make a fair judgment of what this additional complication got me. Not actually building the objects...left me wondering what was going on....and what I did seemed trivial once I got a clue as to what I was...doing."

LIST OF REFERENCES

Barr, A. and Feigenbaum, E.A., *The Handbook of Artificial Intelligence*, Volume 2, pp. 229-234, 291, William Kaufmann, Inc., 1982.

Carlson, P.A., and Crevoisier, M.L., *R-WISE: A Computerized Environment for Tutoring Critical Literacy*, World Conference on Educational Multimedia and Hypermedia, pp. 111-116, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Davis, E., *Representations of Commonsense Knowledge*, pp. 390-391, Morgan Kaufmann Publishers, Inc, 1990.

Elson-Cook, M., Presentation as a panelist for *Authoring for ITS: From minimalist approach to ITS Shells*, conducted at the World Conference on Educational Multimedia and Hypermedia, Vancouver, BC, 1994.

Feifer, R. and Allender, L., *It's Not How Multi the Media, It's How the Media is Used*, World Conference on Educational Multimedia and Hypermedia, pp. 197-202, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Galvin, T.P., and Rowe, N.C., *Using the A* Search Space to Develop a General-Purpose Intelligent Tutoring Shell*, World Conference on Educational Multimedia and Hypermedia, pp. 725, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Gescei, J. and Frasson, C., *SAFARI: an Environment for Creating Tutoring Systems in Industrial Training*, World Conference on Educational Multimedia and Hypermedia, pp. 15-20, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Guralnik, D., and Kass, A., *An Authoring System for Creating Computer-based Role-Performance Trainers*, World Conference on Educational Multimedia and Hypermedia, pp. 235-240, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Heift, T., and McFetridge, P., *The Intelligent Workbook*, World Conference on Educational Multimedia and Hypermedia, pp. 263-268, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Homem de Mello, L.S. and Sanderson, A.C., *A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences*, IEEE Transactions on Robotics and Automation, Volume 7, Issue 2, pp. 228-240, IEEE Computer Society, Spring Field, MD, 1991.

Jones, M.K., Gibbons, A.S., and Varner, D.C., *A Re-Usable Algorithm for Teaching Procedural Skills*, World Conference on Educational Multimedia and Hypermedia, pp. 299-304, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Ki, W., and others, *A Knowledge-Based Multimedia System to Support the Teaching and Learning of Chinese Characters*, World Conference on Educational Multimedia and Hypermedia, pp. 323-328, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Lalonde, W., and Pugh, J., *Subclassing* ≠ *Subtyping* ≠ *Is-a*, Journal for Object-Oriented Programming, pp. 57-62, January, 1991.

McDowell, Perry, Homework Assignment for CS-4310: Advanced Artificial Intelligence, Winter Quarter, Naval Postgraduate School, 1993.

Moreland, R., *On-Line Tutorials Even YOU Can Create! Computer-Based Multimedia Training Slide Shows*, World Conference on Educational Multimedia and Hypermedia, p. 748, Association for the Advancement of Computing in Education, Vancouver, BC, 1994.

Murray, T., Presentation as a panelist for *Authoring for ITS: From minimalist approach to ITS Shells*, conducted at the World Conference on Educational Multimedia and Hypermedia, Vancouver, BC, 1994.

Naval Postgraduate School Report NPSCS-93-009, *Instructions for Use of the METutor Means-Ends Tutoring System*, by N.C. Rowe, pp. 1-7, July, 1993.

Psotka, J., Massey, L.D., and Mutter, S.A., *Intelligent Tutoring Systems: Lessons Learned*, p. 5, Lawrence Erlbaum Associates, 1987.

Rowe, N.C., and Suwono, F., *Aiding Teachers in Constructing Virtual-Reality Tutors*, Fourth Annual Conference on Artificial Intelligence, Simulation, and Planning in High Autonomy Systems, pp. 317-323, Tuscon, AZ, 1993.

Rumbaugh, J., and others, *Object-Oriented Modeling and Design*, pp. 2,57-91, Prentice Hall, 1991.

Sacerdoti, E.D., *The Non-Linear Nature of Plans*, in Readings in Planning, Allen, J., Hendler, J., and Tate, A., eds., pp. 162-170, Morgan Kaufmann Publishers, Inc., 1990.

Seem, Dennis, Homework Assignment for CS-4310: Advanced Artificial Intelligence, Winter Quarter, Naval Postgraduate School, 1992.

Sierra On-Line, Inc., *Leisure Suit Larry 6: Shape Up or Slip Out*, MS-DOS Version 1.0, 1993.

Sleeman, D., *PIXIE: A Shell for Developing Intelligent Tutoring Systems, in n Artificial Intelligence in Education, Volume 1*, pp. 239-265, Ablex Publishing, 1987.

Tenney, Y.J., and Kurland, L.C., *The Development of Troubleshooting Expertise in Radar Mechanics*, in Intelligent Tutoring Systems: Lessons Learned, Massey, L.D., Mutter, S.A., and Psotka, J., eds., pp. 59-84, Lawrence Erlbaum Associated Press, 1987.

U.S. Army Training and Doctrine Command (TRADOC), *FM 25-101*, p. 21, Fort Leavenworth, 1991.

Woolf, B. and others, T*eaching a Complex Industrial Process*, in Artificial Intelligence in Education, Volume 1, pp. 413-427, Ablex Publishing, 1987.

BIBLIOGRAPHY

Allen, J., Hendler, J., and Tate, A., *Readings in Planning*, Morgan Kaufman Publishers, San Mateo, CA, 1990.

Association for the Advancement of Computing in Education, *Proceedings of the World Conference on Educational Multimedia and Hypermedia*, Vancouver, BC, 1994.

Barr, A., and Feigenbaum, E.A., *The Handbook of Artificial Intelligence: Volume 2*, HeurisTech Press, Stanford, CA, 1982.

Davis, E., *Representations of Commonsense Knowledge*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

McGraw, K.L., and Harbison-Briggs, K., *Knowledge Acquisition: Principles and Guidelines*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

Michalski, R.S., Carbonnell, J.G., and Mitchell, T.M., *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, Palo Alto, CA, 1983.

Psotka, J., Massey, L.D., and Mutter, S.A., *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

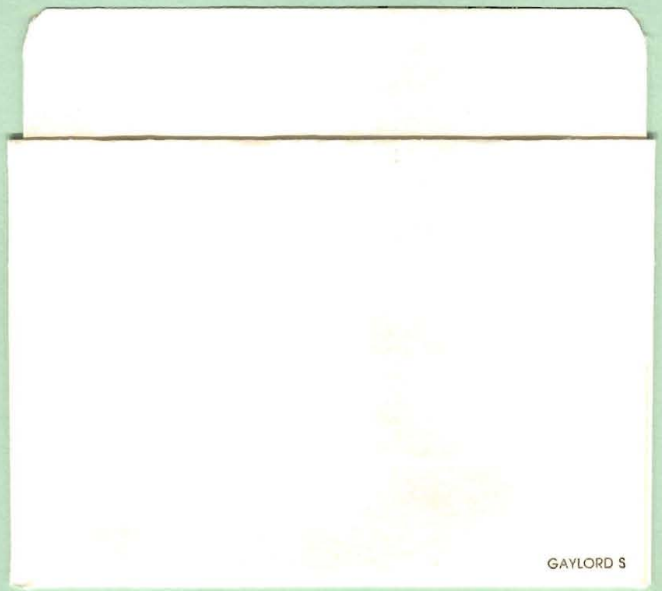
Rumbaugh, J., and others, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Sleeman, D., and Brown, J.S., *Intelligent Tutoring Systems*, Academic Press, 1982.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, California 93943	2
Professor Neil C. Rowe, Code CS/RP Computer Science Department Naval Postgraduate School Monterey, California 93943	5
Professor Timothy M. Shimeall, Code CS/SM Computer Science Department Naval Postgraduate School Monterey, California 93943	1
Professor Man-Tak Shing, Code CS/SH Computer Science Department Naval Postgraduate School Monterey, California 93943	1
Lieutenant Commander John Daley, Code CS/DA Computer Science Department Naval Postgraduate School Monterey, California 93943	1

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101



DUDLEY KNOX LIBRARY
3 2768 00099285 3